BLG 317E
DATABASE SYSTEMS

WEEK 10

Transactions

Transactions

- a group of operations may have to be carried out together
- finishing some operations while failing on others might cause inconsistency
- transaction: a logical unit of work
- no guarantee that multiple operations will all be finished
- at least return to the state before the changes

Transactions

- a group of operations may have to be carried out together
- finishing some operations while failing on others might cause inconsistency
- transaction: a logical unit of work
- no guarantee that multiple operations will all be finished
- at least return to the state before the changes

Example: Money transfer between two bank accounts

- Suppose a bank customer wants to transfer money from one bank account to another account.
- Suppose both accounts belong to the same bank.
- The steps to do that in the application could be:
  - Select two accounts to transfer (From and To)
  - Enter the amount to transfer
  - Click “Transfer” button
  - The “Account from” has the amount reduced
  - The “Account to” has the amount increased

Example: Money transfer between two bank accounts

- Once the customer clicks Transfer button to transfer the money, the balances of both accounts are updated.
- The following two UPDATE statements will transfer $100, from account_id 1 to account_id 2.
- Database table name is ACCOUNT.
- If a computer crash occurs between the two UPDATE statements, the operation is incomplete.

```
UPDATE ACCOUNT
SET balance = balance – 100
WHERE account_id = 1;

UPDATE ACCOUNT
SET balance = balance + 100
WHERE account_id = 2;
```

A Failure Situation

STATEMENTS	SITUATION	RESULT

`UPDATE account`
`SET balance = balance – 100`
`WHERE account_id = 1;`
 	
`UPDATE account`
`SET balance = balance + 100`
`WHERE account_id = 2;`
	• First statement succeeds.
	
	• Second statement fails.
		• Account 1 balance is now $100 less.
		
		• But Account 2 balance remains the same.

- The two statements should be executed together.
- Either both statements run successfully, or both statements don't.
- That is the purpose of a transaction.

Transaction Properties

- Atomicity: all or nothing
- Consistency: move from one consistent state to another
- Isolation: whether operations of an unfinished transaction affect other transactions or not
- Durability: when a transaction is finished,
  its changes are permanent even if there is a system failure

```
UPDATE account
SET balance = balance - 100
WHERE account_id = 1;


UPDATE account
SET balance = balance + 100
WHERE account_id = 2;
```

Succeed or fail altogether!

Transaction Properties

- Atomicity: all or nothing
- Consistency: move from one consistent state to another
- Isolation: whether operations of an unfinished transaction affect other transactions or not
- Durability: when a transaction is finished,
  its changes are permanent even if there is a system failure

```
UPDATE account
SET balance = balance - 100
WHERE account_id = 1;


UPDATE account
SET balance = balance + 100
WHERE account_id = 2;
```

Account1: 500
Account2: 200

state1

Account1: 400
Account2: 300

state2

Transaction Properties

- Atomicity: all or nothing
- Consistency: move from one consistent state to another
- Isolation: whether operations of an unfinished transaction affect other transactions or not
- Durability: when a transaction is finished, its changes are permanent even if there is a system failure

Transaction 1

```
UPDATE account
SET balance = balance - 100
WHERE account_id = 1;
```

Transaction 2

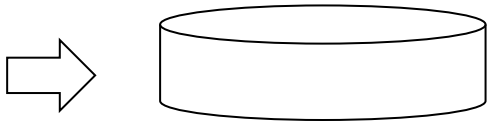
```
select account_id, balance
from account;
```

Transaction Properties

- Atomicity: all or nothing
- Consistency: move from one consistent state to another
- Isolation: whether operations of an unfinished transaction affect other transactions or not
- Durability: when a transaction is finished,
  its changes are permanent even if there is a system failure

```
UPDATE account
SET balance = balance – 100
WHERE account_id = 1;

COMMIT;
```



SQL Transaction Commands

- The followings are Transaction Control Language (TCL) commands for writing transactions in SQL.
  - **START**
  - **COMMIT**
  - **ROLLBACK**
  - **SAVEPOINT**

START Command

- The START command will begin a new transaction.
- There are some differences between DBMSs, for the implementation of transactions.
- In SQL Server, transactions must be defined explicitly.
- In Oracle and MySQL, implicit transactions are enabled automatically.
  - A transaction is started automatically, when a database session is started.
  - Every SQL command (INSERT, DELETE, UPDATE) is auto-commit.
  - Sometimes it may be necessary to begin a new transaction at a specific point.

COMMIT Command

- The Commit command indicates that the transaction has ended successfully and the data should be saved permanently.
- The changes made between the start of the transaction and the COMMIT statement are permanently stored (committed) on the database. Any savepoints created are removed.
- A commit can occur:
  - When you run a DDL statement (e.g. CREATE TABLE)
  - When you exit the IDE and indicate you want to commit the changes
  - When you run a COMMIT command

ROLLBACK Command

- The Rollback command allows to undo (cancel) the changes in a transaction and restore the database to a state before the transaction began.
- It's the opposite to a Commit.
- Just like a commit, a rollback can happen automatically if an error occurs (an implicit rollback) or you can specify it at a certain point (an explicit rollback).
- The changes made by the Update statements are not stored in the database, and the data is reverted back to the way it was before the statements were run.
- SQL commands can be run after the SAVEPOINT command. The transaction can then be rolled back to the savepoint if needed, and the statements before the savepoint are NOT lost.
- A rollback can occur:
  - When an exception happens in your query
  - When the SQL IDE exits
  - When you run a ROLLBACK command

Transaction Example

```
CREATE PROCEDURE pTransferMoney()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
    END;

    START TRANSACTION;

    UPDATE account SET balance = balance - 100
    WHERE account_id = 1;

    UPDATE account SET balance = balance + 100
    WHERE account_id = 2;

    COMMIT;
END;
```

SAVEPOINT Command

- When running a transaction, you can specify a savepoint, which is a point in the transaction that can be used as a point to return to upon a failure.
- If something fails in the transaction after the savepoint, you can rollback to the savepoint and continue the transaction, instead of rolling back the entire transaction.
- SAVEPOINT allows to perform a partial rollback of a transaction.
- To do this, issue a SAVEPOINT statement within the transaction to set a marker.
- To roll back to just that point in the transaction later, use a ROLLBACK statement that names the savepoint.

SQL Transaction Example

```
CREATE TABLE PERSON (id INT);
```

Transaction

```
START TRANSACTION;
INSERT INTO PERSON VALUES(100);
SAVEPOINT savepoint1;

INSERT INTO PERSON VALUES(200);
ROLLBACK TO SAVEPOINT savepoint1;
INSERT INTO PERSON VALUES(300);
COMMIT;
```

INSERT (200) cancelled

Query

```
SELECT * FROM PERSON;
```

Result

```
+------+
|  id  |
+------+
|  100 |
|  300 |
+------+
```

Example: ROLLBACK TO SAVEPOINT

- A savepoint can be created with the SAVEPOINT command.
- SQL commands can be run after the SAVEPOINT command.
- The transaction can then be rolled back to the savepoint if needed, and the statements before the savepoint are NOT lost.

```
START TRANSACTION;
UPDATE account SET balance = balance - 100 WHERE account_id = 1;
UPDATE account SET balance = balance + 100 WHERE account_id = 2;
SAVEPOINT savepoint1;
Other Statements;
ROLLBACK TO SAVEPOINT savepoint1;
COMMIT;
```

When Does a Transaction End

- An SQL transaction can end, either successfully or unsuccessfully, in many situations:
- A COMMIT command is issued
- A ROLLBACK command is issued
- A DDL statement is run
- The SQL IDE is closed
- The connection to the database is lost
- The database encounters an error

Transactions in MySQL

- By default, MySQL runs in autocommit mode, which means that changes made by individual statements are committed to the database immediately to make them permanent.
- Each statement is its own transaction implicitly.
- You can disable autocommit for a transaction by starting a transaction with an explicit START command.
- To perform transactions explicitly, disable autocommit mode.

```
SET autocommit = 0;     Disables

SET autocommit = 1;     Enables
```

Recovery

- consider a system failure during a transaction
- buffer cache has not been flushed to the disk
- how to guarantee durability?
- derive the data from other sources in the system

Recovery

- consider a system failure during a transaction
- buffer cache has not been flushed to the disk
- how to guarantee durability?
  - derive the data from other sources in the system

Transaction Log

- log: values of every affected tuple
  before and after every operation
- write-ahead log rule:
  log must be flushed before the transaction is committed
- accessing records in the log is sequential by nature

Transaction Log

- log: values of every affected tuple
  before and after every operation
- write-ahead log rule:
  log must be flushed before the transaction is committed
- accessing records in the log is sequential by nature

Checkpoints

- create checkpoints at certain intervals:
- flush buffer cache to the physical medium
- note the checkpoint
- note the continuing transactions

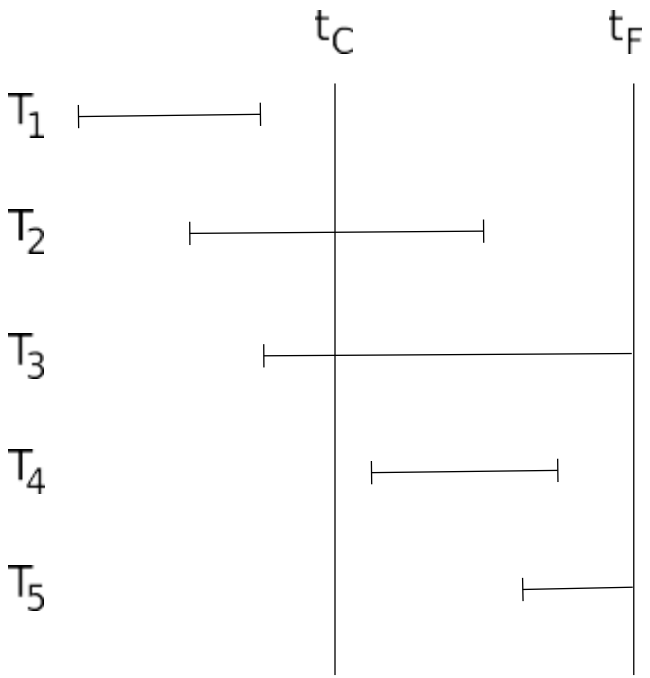
Recovery Lists

- after the failure: which transactions will be undone, which transactions will be made permanent?
- create two lists: *undo* (U), *redo* (R)
- t_C: last checkpoint in the log
- add the transactions which are active at t_C to U
- scan records from t_C to end of log
- add any starting transaction to U
- move any finishing transaction to R

Recovery Lists

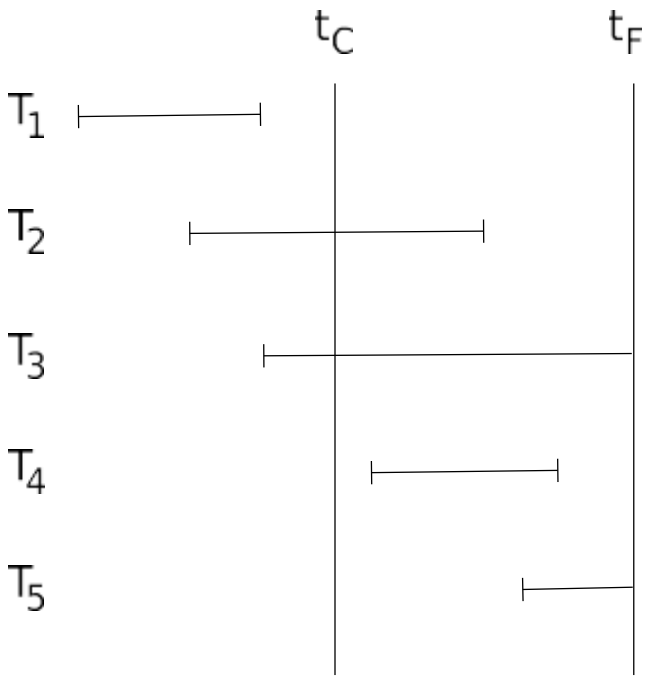
- after the failure: which transactions will be undone, which transactions will be made permanent?
- create two lists: *undo* (U), *redo* (R)
- t_C: last checkpoint in the log
- add the transactions which are active at t_C to U
- scan records from t_C to end of log
- add any starting transaction to U
- move any finishing transaction to R

Recovery Example



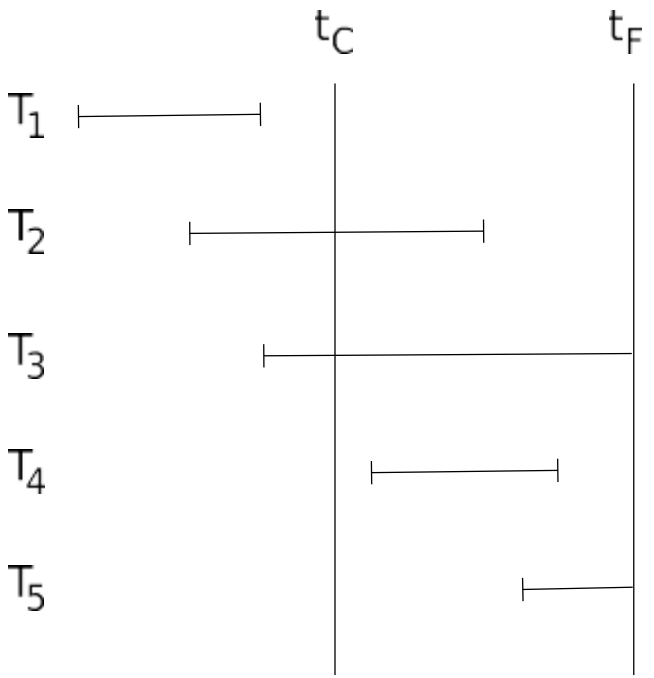
- t_C:
  $U = [T_2, T_3]$ $R = []$
- T_4 started:
  $U = [T_2, T_3, T_4]$ $R = []$
- T_2 finished:
  $U = [T_3, T_4]$ $R = [T_2]$
- T_5 started:
  $U = [T_3, T_4, T_5]$ $R = [T_2]$
- T_4 finished:
  $U = [T_3, T_5]$ $R = [T_2, T_4]$

Recovery Example



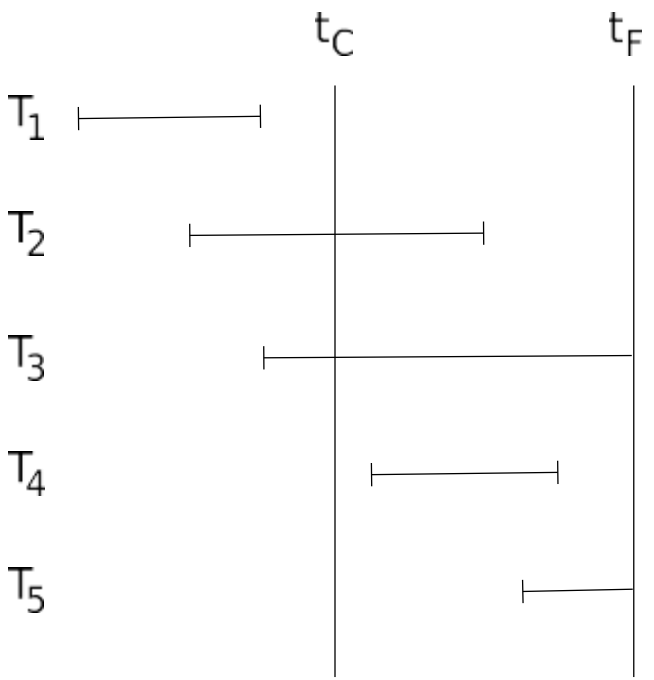
- t_C:
  $U = [T_2, T_3]$ $R = []$
- T_4 started:
  $U = [T_2, T_3, T_4]$ $R = []$
- T_2 finished:
  $U = [T_3, T_4]$ $R = [T_2]$
- T_5 started:
  $U = [T_3, T_4, T_5]$ $R = [T_2]$
- T_4 finished:
  $U = [T_3, T_5]$ $R = [T_2, T_4]$

Recovery Example



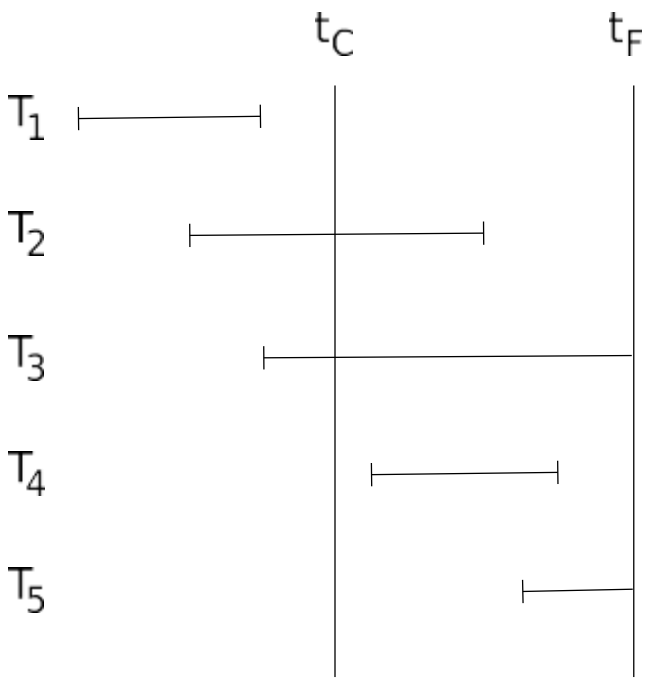
- t_C:
  $U = [T_2, T_3]$ $R = []$
- T_4 started:
  $U = [T_2, T_3, T_4]$ $R = []$
- T_2 finished:
  $U = [T_3, T_4]$ $R = [T_2]$
- T_5 started:
  $U = [T_3, T_4, T_5]$ $R = [T_2]$
- T_4 finished:
  $U = [T_3, T_5]$ $R = [T_2, T_4]$

Recovery Example



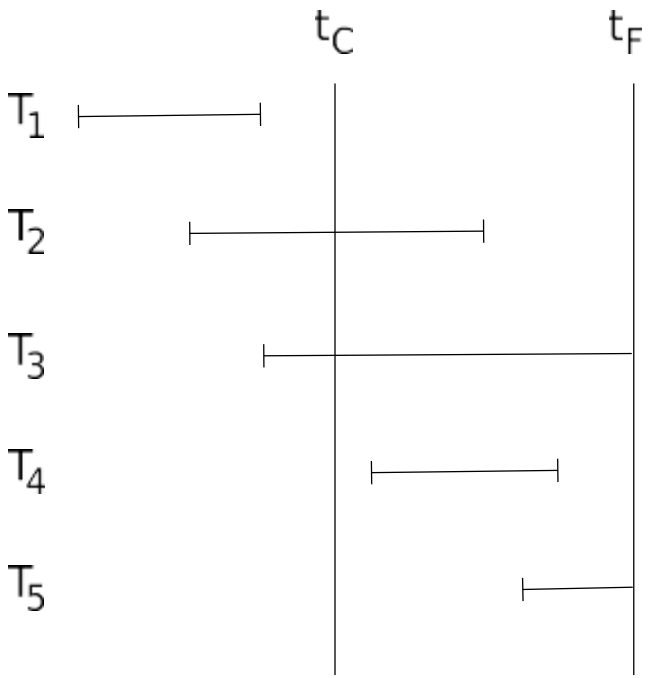
- t_C:
  $U = [T_2, T_3]$ $R = []$
- T_4 started:
  $U = [T_2, T_3, T_4]$ $R = []$
- T_2 finished:
  $U = [T_3, T_4]$ $R = [T_2]$
- T_5 started:
  $U = [T_3, T_4, T_5]$ $R = [T_2]$
- T_4 finished:
  $U = [T_3, T_5]$ $R = [T_2, T_4]$

Recovery Example



- t_C:
  $U = [T_2, T_3]$ $R = []$
- T_4 started:
  $U = [T_2, T_3, T_4]$ $R = []$
- T_2 finished:
  $U = [T_3, T_4]$ $R = [T_2]$
- T_5 started:
  $U = [T_3, T_4, T_5]$ $R = [T_2]$
- T_4 finished:
  $U = [T_3, T_5]$ $R = [T_2, T_4]$

Recovery Example



- t_C:
  $U = [T_2, T_3]$ $R = []$
- T_4 started:
  $U = [T_2, T_3, T_4]$ $R = []$
- T_2 finished:
  $U = [T_3, T_4]$ $R = [T_2]$
- T_5 started:
  $U = [T_3, T_4, T_5]$ $R = [T_2]$
- T_4 finished:
  $U = [T_3, T_5]$ $R = [T_2, T_4]$

Recovery Process

- scan records backwards from end of log
- undo the changes made by the transactions in *U*
- scan records forwards
- redo the changes made by the transactions in *R*

Transactions in Distributed Environments

Two-Phase Commit

- different source managers
- different undo / redo mechanisms
- modifications on data on different source managers
- either commit in all or rollback in all
- coordinator

Protocol

- coordinator → participants:
  prepare and report back the result
- upon prepare command, participants:
  flush all data regarding the transaction
- all participants succeeded: success
- otherwise: failure
- if success, coordinator → participants: commit
- if failure, coordinator → participants: rollback

Protocol

- coordinator → participants:
  prepare and report back the result
- upon prepare command, participants:
  flush all data regarding the transaction
- all participants succeeded: success
- otherwise: failure
- if success, coordinator → participants: commit
- if failure, coordinator → participants: rollback

Protocol

- coordinator → participants:
  prepare and report back the result
- upon prepare command, participants:
  flush all data regarding the transaction
- all participants succeeded: success
- otherwise: failure
- if success, coordinator → participants: commit
- if failure, coordinator → participants: rollback

Concurrency

Concurrency

- problems that might arise due to simultaneuous transactions:
  - lost update
  - uncommitted dependency
  - inconsistent analysis
  - phantom reads

Lost Update

Transaction A	Transaction B
…	…
RETRIEVE p	…
…	…
…	RETRIEVE p
…	…
UPDATE p	…
…	…
…	UPDATE p
…	…

Lost Update

Transaction A	Transaction B
...	...
RETRIEVE p	...
...	...
...	RETRIEVE p
...	...
UPDATE p	...
...	...
...	UPDATE p
...	...

Lost Update

Transaction A	Transaction B
…	…
RETRIEVE p	…
…	…
…	RETRIEVE p
…	…
UPDATE p	…
…	…
…	UPDATE p
…	…

Lost Update

Transaction A	Transaction B
...	...
RETRIEVE p	...
...	...
...	RETRIEVE p
...	...
UPDATE p	...
...	...
...	UPDATE p
...	...

Lost Update

Transaction A	Transaction B
…	…
RETRIEVE p	…
…	…
…	RETRIEVE p
…	…
UPDATE p	…
…	…
…	UPDATE p
…	…

Dirty Read (Uncommitted Dependency)

Transaction A	Transaction B
…	…
…	UPDATE p
…	…
RETRIEVE p	…
…	…
…	ROLLBACK
…	

Dirty Read (Uncommitted Dependency)

Transaction A	Transaction B
...	...
...	UPDATE p
...	...
RETRIEVE p	...
...	...
...	ROLLBACK
...	

Dirty Read (Uncommitted Dependency)

Transaction A	Transaction B
…	…
…	UPDATE p
…	…
RETRIEVE p	…
…	…
…	ROLLBACK
…	

Dirty Read (Uncommitted Dependency)

Transaction A	Transaction B
…	…
…	UPDATE p
…	…
RETRIEVE p	…
…	…
…	ROLLBACK
…	

Non-repeatable Read

Transaction A	Transaction B
…	…
RETRIEVE p	…
…	…
…	UPDATE p
…	…
…	COMMIT
…	
RETRIEVE p	

Non-repeatable Read

Transaction A	Transaction B
…	…
RETRIEVE p	…
…	…
…	UPDATE p
…	…
…	COMMIT
…	
RETRIEVE p	

Non-repeatable Read

Transaction A	Transaction B
…	…
RETRIEVE p	…
…	…
…	UPDATE p
…	…
…	COMMIT
…	
RETRIEVE p	

Non-repeatable Read

Transaction A	Transaction B
…	…
RETRIEVE p	…
…	…
…	UPDATE p
…	…
…	COMMIT
…	
RETRIEVE p	

Inconsistent Analysis

- compute sum of accounts: acc1=40, acc2=50, acc3=30

Transaction A	Transaction B
...	...
RETRIEVE acc1 (40)	...
RETRIEVE acc2 (90)	...
...	...
...	UPDATE acc3 (30 → 20)
...	UPDATE acc1 (40 → 50)
...	COMMIT
...	...
RETRIEVE acc3 (110)	
...	

Inconsistent Analysis

- compute sum of accounts: acc1=40, acc2=50, acc3=30

Transaction A	Transaction B
...	...
RETRIEVE acc1 (40)	...
RETRIEVE acc2 (90)	...
...	...
...	UPDATE acc3 (30 → 20)
...	UPDATE acc1 (40 → 50)
...	COMMIT
...	...
RETRIEVE acc3 (110)	
...	

Inconsistent Analysis

- compute sum of accounts: acc1=40, acc2=50, acc3=30

Transaction A	Transaction B
…	…
RETRIEVE acc1 (40)	…
RETRIEVE acc2 (90)	…
…	…
…	UPDATE acc3 (30 → 20)
…	UPDATE acc1 (40 → 50)
…	COMMIT
…	…
RETRIEVE acc3 (110)	
…	

Inconsistent Analysis

- compute sum of accounts: acc1=40, acc2=50, acc3=30

Transaction A	Transaction B
…	…
RETRIEVE acc1 (40)	…
RETRIEVE acc2 (90)	…
…	…
…	UPDATE acc3 (30 → 20)
…	UPDATE acc1 (40 → 50)
…	COMMIT
…	…
RETRIEVE acc3 (110)	
…	

Inconsistent Analysis

- compute sum of accounts: acc1=40, acc2=50, acc3=30

Transaction A	Transaction B
…	…
RETRIEVE acc1 (40)	…
RETRIEVE acc2 (90)	…
…	…
…	UPDATE acc3 (30 → 20)
…	UPDATE acc1 (40 → 50)
…	COMMIT
…	…
RETRIEVE acc3 (110)	
…	

Phantom Reads

- phantom: when the query is executed again, new tuples appear

Example

- A computes the average of a customer's account balances:

$$\frac{100+100+100}{3} = 100$$

- B creates new account (200) for the same customer
- A computes again:

$$\frac{100+100+100+200}{4} = 125$$

Phantom Reads

- phantom: when a select query is executed again, new tuples appear

Example

- A computes the average of a customer's account balances:

$$\frac{100+100+100}{3} = 100$$

- B creates new account (200) for the same customer
- A computes again:

$$\frac{100+100+100+200}{4} = 125$$

Conflicts

- A reads, B reads: no problem
- A reads, B writes: non-repeatable read (inconsistent analysis)
- A writes, B reads: dirty read (uncommitted dependency)
- A writes, B writes: dirty write (lost update)

Locks

- let transactions lock the tuples they work on
  - shared lock (S)
  - exclusive lock (X)

Lock Requests

lock type compatibility matrix

	S	X
-	Y	Y
S	Y	N
X	N	N

- existing shared lock: only shared lock requests are allowed
- existing exclusive lock: all lock requests are denied

Locking Protocol

- transaction requests lock depending on operation
- promote a shared lock to an exclusive lock
- if request denied, it starts waiting
- it continues when the transaction that holds the lock releases it
- starvation

Lost Update

Transaction A	Transaction B
...	...
RETRIEVE p (S+)	...
...	...
...	RETRIEVE p (S+)
...	...
UPDATE p (X-)	...
wait	...
wait	UPDATE p (X-)
wait	wait

Lost Update

Transaction A	Transaction B
...	...
RETRIEVE p (S+)	...
...	...
...	RETRIEVE p (S+)
...	...
UPDATE p (X-)	...
wait	...
wait	UPDATE p (X-)
wait	wait

Lost Update

Transaction A	Transaction B
…	…
RETRIEVE p (S+)	…
…	…
…	RETRIEVE p (S+)
…	…
UPDATE p (X-)	…
wait	…
wait	UPDATE p (X-)
wait	wait

Lost Update

Transaction A	Transaction B
…	…
RETRIEVE p (S+)	…
…	…
…	RETRIEVE p (S+)
…	…
UPDATE p (X-)	…
wait	…
wait	UPDATE p (X-)
wait	wait

Lost Update

Transaction A	Transaction B
…	…
RETRIEVE p (S+)	…
…	…
…	RETRIEVE p (S+)
…	…
UPDATE p (X-)	…
wait	…
wait	UPDATE p (X-)
wait	wait

Dirty Read (Uncommitted Dependency)

Transaction A	Transaction B
...	...
...	UPDATE p (X+)
...	...
RETRIEVE p (S-)	...
wait	...
wait	ROLLBACK
RETRIEVE p (S+)	
...	

Dirty Read (Uncommitted Dependency)

Transaction A	Transaction B
...	...
...	UPDATE p (X+)
...	...
RETRIEVE p (S-)	...
wait	...
wait	ROLLBACK
RETRIEVE p (S+)	
...	

Dirty Read (Uncommitted Dependency)

Transaction A	Transaction B
...	...
...	UPDATE p (X+)
...	...
RETRIEVE p (S-)	...
wait	...
wait	ROLLBACK
RETRIEVE p (S+)	
...	

Dirty Read (Uncommitted Dependency)

Transaction A	Transaction B
...	...
...	UPDATE p (X+)
...	...
RETRIEVE p (S-)	...
wait	...
wait	ROLLBACK
RETRIEVE p (S+)	
...	

Non-repeatable Read

Transaction A	Transaction B
...	...
RETRIEVE p (S+)	...
...	...
...	UPDATE p (X-)
...	wait
...	wait
...	wait
RETRIEVE p (S+)	wait

Non-repeatable Read

Transaction A	Transaction B
...	...
RETRIEVE p (S+)	...
...	...
...	UPDATE p (X-)
...	wait
...	wait
...	wait
RETRIEVE p (S+)	wait

Non-repeatable Read

Transaction A	Transaction B
...	...
RETRIEVE p (S+)	...
...	...
...	UPDATE p (X-)
...	wait
...	wait
...	wait
RETRIEVE p (S+)	wait

Non-Repeatable Read (Inconsistent Analysis)

Transaction A	Transaction B
…	…
RETRIEVE acc1 (S+)	…
RETRIEVE acc2 (S+)	…
…	…
…	UPDATE acc3 (X+)
…	UPDATE acc1 (X-)
…	wait
RETRIEVE acc3 (S-)	wait
wait	wait

Non-Repeatable Read (Inconsistent Analysis)

Transaction A	Transaction B
...	...
RETRIEVE acc1 (S+)	...
RETRIEVE acc2 (S+)	...
...	...
...	UPDATE acc3 (X+)
...	UPDATE acc1 (X-)
...	wait
RETRIEVE acc3 (S-)	wait
wait	wait

Non-Repeatable Read (Inconsistent Analysis)

Transaction A	Transaction B
...	...
RETRIEVE acc1 (S+)	...
RETRIEVE acc2 (S+)	...
...	...
...	UPDATE acc3 (X+)
...	UPDATE acc1 (X-)
...	wait
RETRIEVE acc3 (S-)	wait
wait	wait

Non-Repeatable Read (Inconsistent Analysis)

Transaction A	Transaction B
...	...
RETRIEVE acc1 (S+)	...
RETRIEVE acc2 (S+)	...
...	...
...	UPDATE acc3 (X+)
...	UPDATE acc1 (X-)
...	wait
RETRIEVE acc3 (S-)	wait
wait	wait

Non-Repeatable Read (Inconsistent Analysis)

Transaction A	Transaction B
...	...
RETRIEVE acc1 (S+)	...
RETRIEVE acc2 (S+)	...
...	...
...	UPDATE acc3 (X+)
...	UPDATE acc1 (X-)
...	wait
RETRIEVE acc3 (S-)	wait
wait	wait

Non-Repeatable Read (Inconsistent Analysis)

Transaction A	Transaction B
...	...
RETRIEVE acc1 (S+)	...
RETRIEVE acc2 (S+)	...
...	...
...	UPDATE acc3 (X+)
...	UPDATE acc1 (X-)
...	wait
RETRIEVE acc3 (S-)	wait
wait	wait

Deadlock

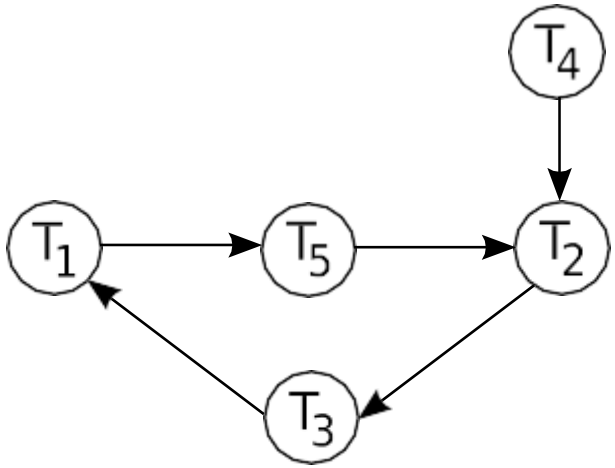
- deadlock: transactions waiting for each other to release the locks
- often happens between two transactions

Approaches:

- detect and solve
- prevent

Deadlock Resolution

Example



- wait graph
- choose a victim and kill it

Preventing Deadlocks

- every transaction has a starting timestamp
- if A's request conflicts with B's lock:
  - wait-die: A waits if older than B;
    Otherwise, A dies. A is rolled back and restarted
  - wound-wait: A waits if younger than B;
    Otherwise, it wounds B, B is rolled back and restarted.
- timestamp of restarted transaction is not changed

Preventing Deadlocks

- every transaction has a starting timestamp
- if A's request conflicts with B's lock:
  - wait-die: A waits if older than B;
    Otherwise, A dies. A is rolled back and restarted
  - wound-wait: A waits if younger than B;
    Otherwise, it wounds B, B is rolled back and restarted.
- timestamp of restarted transaction is not changed

Preventing Deadlocks

- every transaction has a starting timestamp
- if A's request conflicts with B's lock:
  - wait-die: A waits if older than B;
    Otherwise, A dies. A is rolled back and restarted
  - wound-wait: A waits if younger than B;
    Otherwise, it wounds B, B is rolled back and restarted.
- timestamp of restarted transaction is not changed

Transaction Isolation Levels

- Isolation is one of the ACID transaction properties.
- Isolation property determines how transactions interact with data from other transactions.
- Different isolation levels allow or prevent the various Read Problems that can occur when different transactions run simultaneously.
- To deal with Read Problems, DBMS supports four transaction isolation levels.
- These levels determine which modifications made by one transaction can be seen by other transactions that execute at the same time.
  - Read Uncommitted
  - Read Committed
  - Repeatable Reads
  - Serializable

Transaction Isolation Levels

Level	Description
READ UNCOMMITTED	• A transaction can see modifications made by other transactions even before they have been committed.
READ COMMITTED	• A transaction can see modifications made by other transactions only if they have been committed.
REPEATABLE READ	• If a transaction reads the data in a row twice, the result is repeatable.That is, it gets the same result each time, even if other transactions have changed the row. • A non-repeatable read is where a transaction retrieves data twice for a row, but the data for a row is different on each retrieval. • It's different then a Dirty Read in that the second transaction in a Non-Repeatable Read is committed, but the second transaction in a Dirty Read is not committed.

Transaction Isolation Levels

Level	Description
SERIALIZABLE	• This isolation level is similar to REPEATABLE READ but isolates transactions more completely. • Rows examined by one transaction cannot be modified by other transactions until the first transaction completes (range lock). • This enables one transaction to read rows and at the same time prevent them from being modified by other transactions until it is done with them.

"SET TRANSACTION ISOLATION LEVEL" SQL Command

- You can set the transaction level in the database so it's different to the default.

```
SET TRANSACTION ISOLATION LEVEL <level-name>;
```

- **Example:** The command below will set the transaction isolation level in the database to Read Uncommitted.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

Isolation Level : Read Uncommitted

- It is the lowest level of isolation.
- With this isolation level, transactions can read uncommitted changes from other transactions.
- This means that the following anomalies may still happen:
  - Dirty reads
  - Lost updates
  - Non-repeatable reads
  - Phantom reads

Isolation Level : Read Committed

- With this isolation level, transactions can only read data that has been committed.
- It prevents transactions from seeing changes made by dirty reads (data from uncommitted transactions).
- This means that the following anomalies may still happen:
  - Lost updates
  - Non-repeatable reads
  - Phantom reads

Isolation Level : Repeatable Reads

- With this isolation level, a transaction holds read and write locks on any rows it references.
- This prevents Non-Repeatable Reads from occurring because data locked by a transaction during a SELECT cannot be updated by another transaction.
- It also means Lost Updates and Dirty Reads cannot occur.
- The following anomaly may still happen:
  - Phantom reads

Isolation Level : Serializable Reads

- It is the highest level of isolation.
- With this isolation level, a transaction holds read and write locks on any rows it references.
- It also acquires a “range lock” if a WHERE clause is used on a range so that Phantom Reads are avoided.
- This isolation level prevents all kind of read problems (dirty reads, lost updates, non-repeatable reads, and phantom reads) from occurring.

Serializability

- *serial execution*: a transaction starts only after another is finished
- serializable: result of concurrent execution is always the same as one of the serial executions

Example

- $x = 10$
- transaction A: $x = x + 1$
- transaction B: $x = 2 * x$
- first A, then B: $x = 22$
- first B, then A: $x = 21$

Serializability

- *serial execution*: a transaction starts only after another is finished
- serializable: result of concurrent execution is always the same as one of the serial executions

Example

- $x = 10$
- transaction A: $x = x + 1$
- transaction B: $x = 2 * x$
- first A, then B: $x = 22$
- first B, then A: $x = 21$

Serializability

- *serial execution*: a transaction starts only after another is finished
- serializable: result of concurrent execution is always the same as one of the serial executions

Example

- $x = 10$
- transaction A: $x = x + 1$
- transaction B: $x = 2 * x$
- first A, then B: $x = 22$
- first B, then A: $x = 21$

Serializability

- *serial execution*: a transaction starts only after another is finished
- serializable: result of concurrent execution is always the same as one of the serial executions

Example

- $x = 10$
- transaction A: $x = x + 1$
- transaction B: $x = 2 * x$
- first A, then B: $x = 22$
- first B, then A: $x = 21$

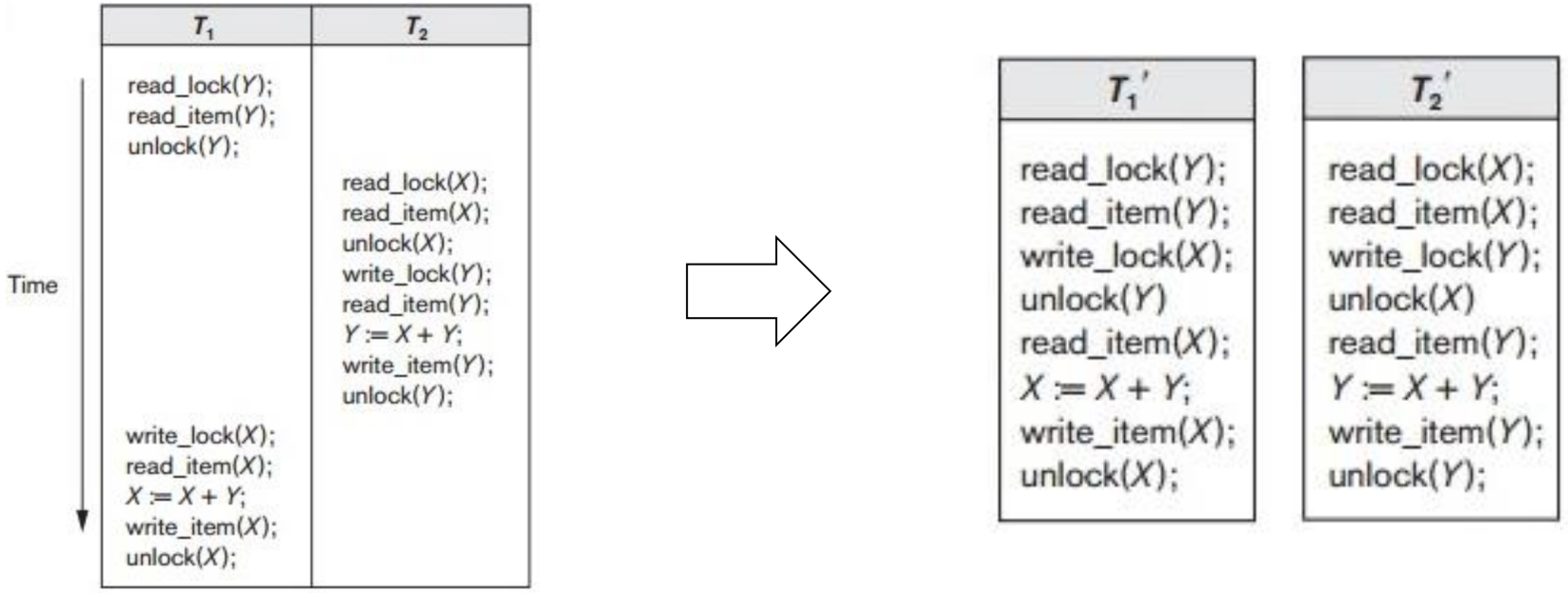
Two-Phase Locking Protocol

- two-phase locking:
  after any lock is released, no more new lock requests
- expansion phase: gather locks
- contraction phase: release locks
- strict two-phase locking:
  all locks are released at the end of the transaction
- *If all transactions obey the two-phase locking protocol, all concurrent executions are serializable.*

Two-Phase Locking Protocol

- two-phase locking:
  after any lock is released, no more new lock requests
- expansion phase: gather locks
- contraction phase: release locks
- two-phase strict locking:
  all locks are released at the end of the transaction
- *If all transactions obey the two-phase locking protocol, all concurrent executions are serializable.*

2PL Example


T_1
T_2
read_lock(Y);
read_item(Y);
unlock(Y);
read_lock(X);
read_item(X);
unlock(X);
write_lock(Y);
read_item(Y);
Y := X + Y;
write_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
X := X + Y;
write_item(X);
unlock(X);
Time
T_1'
read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
X := X + Y;
write_item(X);
unlock(X);
T_2'
read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
Y := X + Y;
write_item(Y);
unlock(Y);

Transaction Isolation Levels and Read Problems

ISOLATION LEVEL	Default Level for DBMS	READ PROBLEM SITUATIONS			
		Dirty Read	Lost Update	Non Repeatable Read	Phantom Read
READ UNCOMMITTED		Yes	Yes	Yes	Yes
READ COMMITTED	Oracle and Microsoft SQL Server	No	Yes	Yes	Yes
REPEATABLE READS	MySQL	No	No	No	Yes
SERIALIZABLE	PostgreSQL	No	No	No	No