

3.0

Concurrency Problems and Isolation Levels

Reading: [KI], chapter 7; [Ha] chapter 9; [Me] chapter 4

ACID

Atomicity:	All-or-Nothing Principle	ensures either all operations within a transaction are successfully completed and committed, or none of them take effect.
Consistency:	All invariants are kept	Referential integrity, primary keys (PK), foreign keys (FK), check constraints, trigger constraints and semantic constraints are called invariants—rules that must not change. We want to work with these invariants. When all invariants are maintained, the system ensures consistency.
Isolation:	Transactions do not interfere with each other	They are executed in a way that makes them appear sequential, even though they actually run in parallel.
Durability:	writes are persistent	whatever you do as a write operation it will be persistent. It is persistent as soon as it's written back to disk. This is sort of persistent storage as long as it's in main memory or cache memory. It is not persistent as soon as system crashes. So durability is about data loss not about whether the data is correct. We can lose correct or incorrect data. Consistency means that data needs to be correct and durability means that correct data needs to be saved.

Durability

Begin;

Begin → this is acknowledgement, there is no semicolon so postgres acknowledges that transaction has begun.

update student set s_balance = s_balance - 3 where s_username= 'Rose';

update 0 1

update teacher set t_payment = t_payment + 3 where t_id = 1;

update 0 1

select t_payment from teacher where t_id = 1;

INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
VALUES (1, '2024-03-03 05:22:12.000000', 'Rose', 'EN');

INSERT 0 1

INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
VALUES (1, '2024-03-04 05:22:12.000000', 'Rose', 'EN');

INSERT 0 1

INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
VALUES (1, '2024-03-05 05:22:12.000000', 'Rose', 'EN');

INSERT 0 1

Time 1 this is wrong because postgres does not even know that transaction comes to an end(does not know if it's about to commit). it is before the commit command of application code. so of course transaction could still go on with any number of operations.

Commit;

Time 2 This is correct. Commit is already issued and transaction is over. When finishing the transaction this means that this is a signal for postgres that it has to make it durable

Commit this is just acknowledging client that everything is done.

Time 3

So time 3 is wrong because everything is over and there is nothing to do

At which point in time do the writes to disk have to take place, time1, **time 2** or time 3?

Durability

here we do not have any acknowledgements since this is just our code

Begin;

```
update student set s_balance = s_balance - 3 where s_username= 'Rose';
```

```
update teacher set t_payment = t_payment + 3 where t_id = 1;
```

```
select t_payment from teacher where t_id = 1;
```

```
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
```

```
VALUES (1, '2024-03-03 05:22:12.000000', 'Rose', 'EN');
```

```
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
```

```
VALUES (1, '2024-03-04 05:22:12.000000', 'Rose', 'EN');
```

```
INSERT INTO lesson(t_id, lesson_time, s_username, subjectcode)
```

```
VALUES (1, '2024-03-05 05:22:12.000000', 'Rose', 'EN');
```

Commit;

when application code says commit then postgres has to write persistently meaning writing out to disk.

how many files does postgres need to access?

heap file page of student table
heap file page of teacher table
heap file page of lesson table
Btree of lesson table

heap file pages are pages where our data is.

if we do inserts and since we usually have PK we need to access Btree

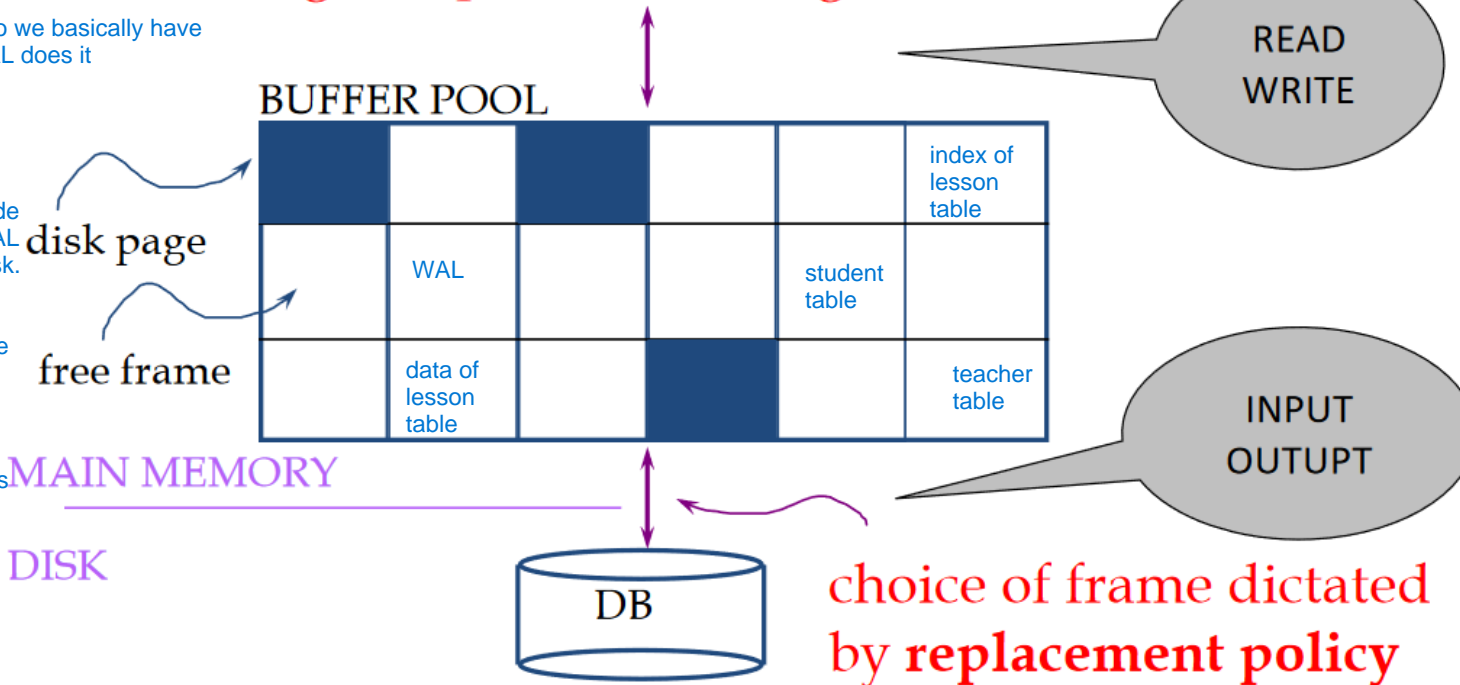
when we talk about loading into the buffer then first we load btree of student and then find the correct page and load it, then we load btree of teacher and data of teacher, then btree of lesson and data of lesson. So 6 buffer pages will be filled but the question is what do we have to write back. We didn't change the index of student and teacher, so we don't have to write them back. We only changed data of student and teacher, index of lesson and data of lesson so out of 6 that we transported we only have to write 4 back.

Buffer Management in a DBMS

for each write operation one record in this WAL is created. it's called xlog. so update, insert, delete all write operations are recorded here. So we basically have the update twice. WAL does it sequentially.

Page Requests from Higher Levels

When application code says commit only WAL page is written on disk. So then in case of a crash we can simply restore the whole database by loading our WAL from disk. So instead of 4 pages we have 1, where we have sequential access which is much faster

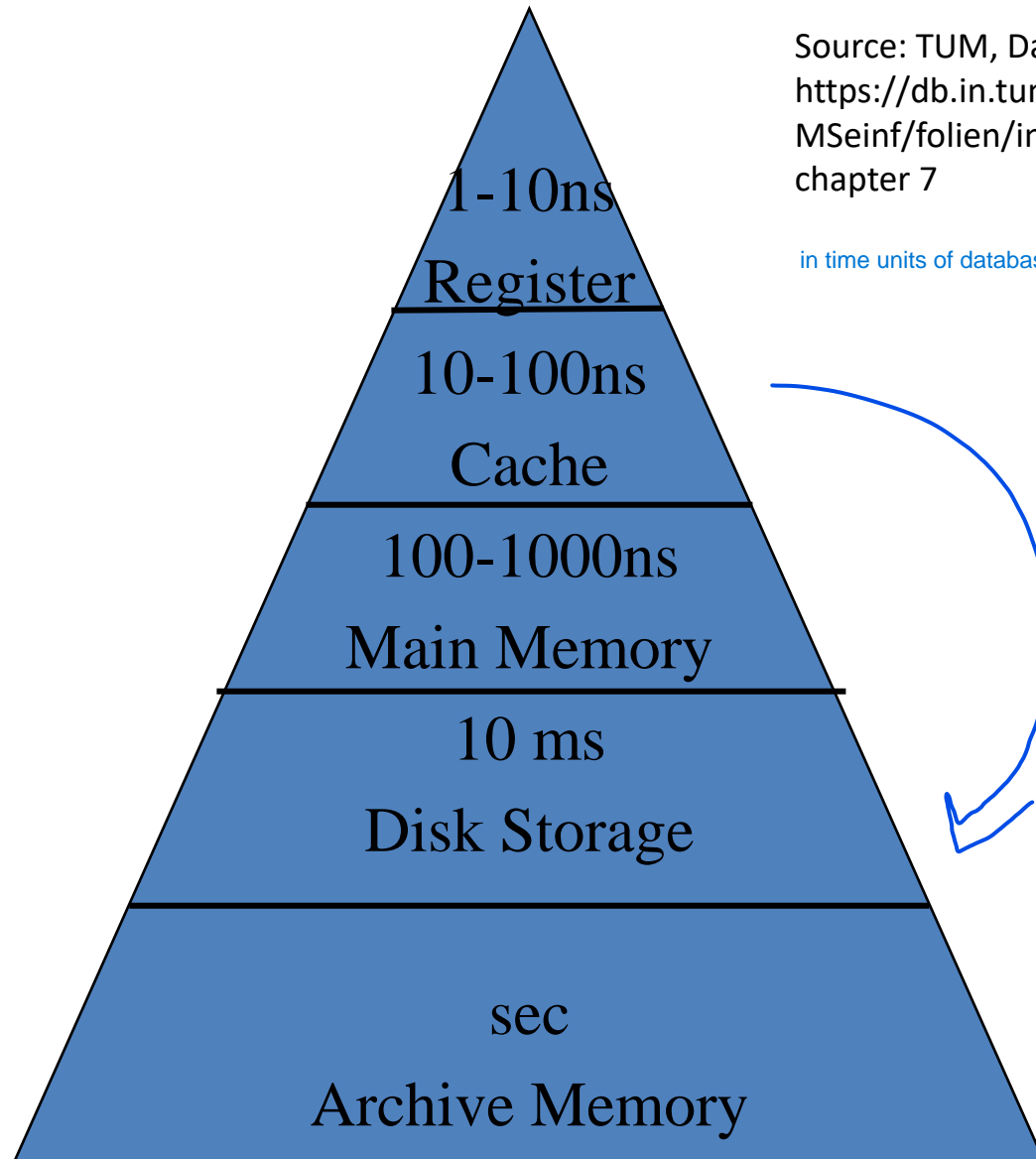


- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained

Access Times

Source: TUM, Database Systems
<https://db.in.tum.de/teaching/bookDB/MSeinf/fohlen/index.shtml?lang=de>,
 chapter 7

in time units of database this takes horribly long



Brain (1min)

Room / building
(10 min)

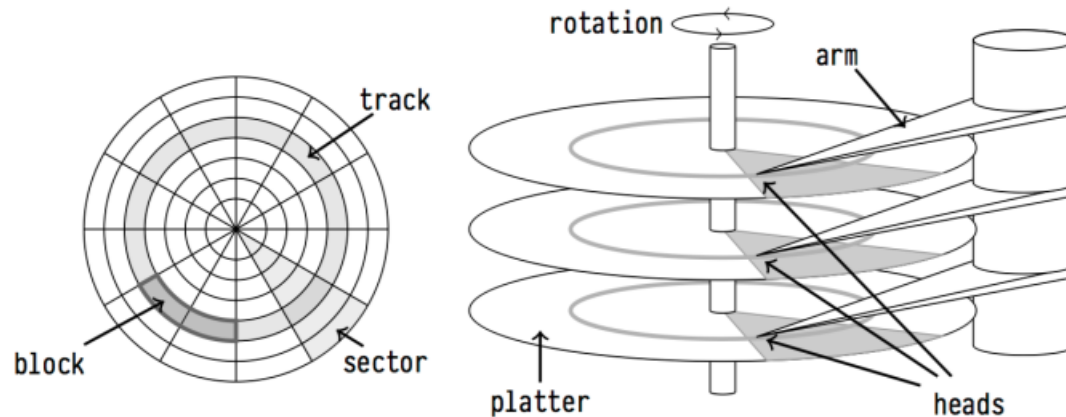
Munich (1.5h)

Pluto (2 years)

so in our case above based on
 this slide we need to
 go to pluto 4 times which takes
 a lot of time

Andromeda
(2000 years)

HDDs: Tracks, Sectors, Blocks



- ❶ **Seek** Stepper motor positions array of R/W heads over wanted **track**.
- ❷ **Rotate** Wait for wanted **sector** of blocks to rotate under R/W heads.
- ❸ **Transfer** Activate one head to read/write **block** data.

Write-Ahead-Log (WAL)

Postgres Documentation:

Write-Ahead Logging (WAL) is a standard method for ensuring data integrity. A detailed description can be found in most (if not all) books about transaction processing. Briefly, WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, after WAL records describing the changes have been flushed to permanent storage. If we follow this procedure, we do not need to flush data pages to disk on every transaction commit,

Using WAL results in a significantly reduced number of disk writes, because only the WAL file needs to be flushed to disk to guarantee that a transaction is committed, rather than every data file changed by the transaction.

so postgres can commit a transaction even before the pages themselves are written back.

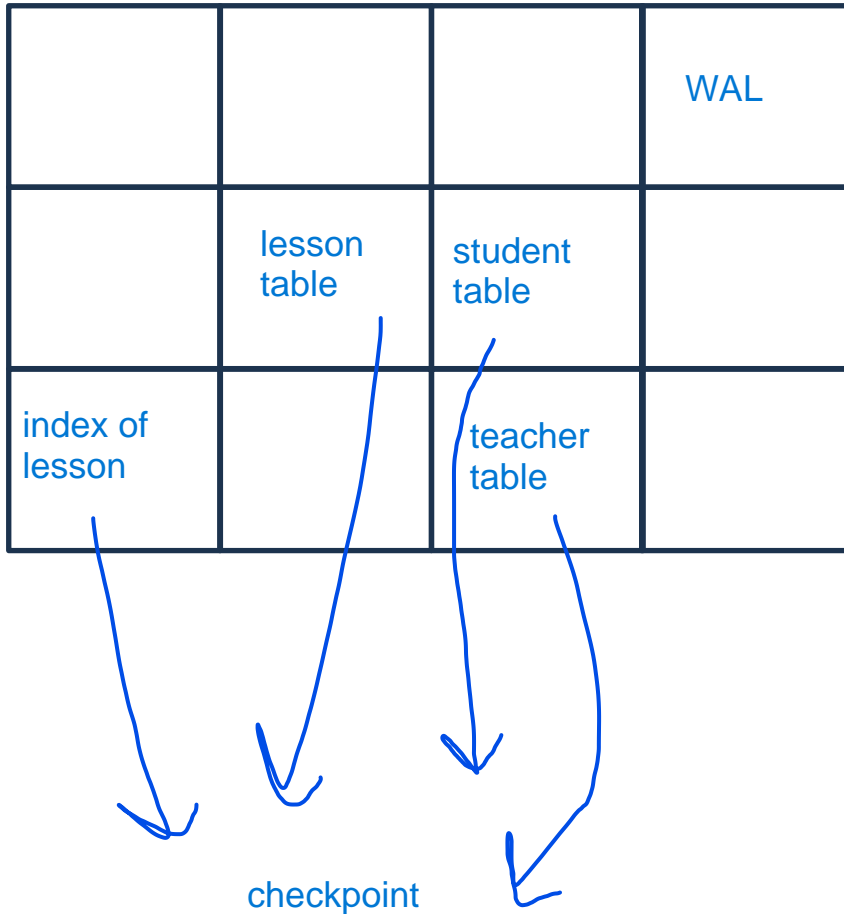
after commit only WAL is flushed

this accelerates our database by factor of however many disk pages we have to write back

We can also decrease the flushing of the WAL

with WAL we get many parameters that either favor performance or consistency(prevention of data loss)

Buffer Pool



1. Update of data page – heapfile student
2. write WAL record to WAL buffer page
3. Update of data page – heapfile teacher
4. write WAL record to WAL buffer page
5. Insert into data page – heapfile lesson
6. Insert into index page – index lesson
7. write WAL records to WAL buffer page
8. Insert into data page – heapfile lesson
9. Insert into index page – index lesson
10. write WAL records to WAL buffer page
11. Insert into data page – heapfile lesson
12. Insert into index page – index lesson
13. write WAL records to WAL buffer page
14. Commit;
15. Flush WAL buffer page to disk

Write-Ahead-Log (WAL)

WALs

- are an additional data-structure on disk
- append-only files (only sequential writes)
- contain an xlog record with a LSN (LogSequenceNumber) for each write operation

each of these xlog records contains 1 write operation and a sequence number

- used for recovery
- speed up of transaction commit

-- returns wal segment files

```
SELECT * FROM pg_ls_waldir();
```

-- returns current LSN

```
SELECT pg_current_wal_lsn();
```

Checkpoints

At checkpoints

- data and index pages are written back to disk.
- WAL and data / index pages are synchronized.
- Happens automatically in regular (configurable) intervals.
- can be run anytime.

we always see checkpoints which are automated points in time when postgres also writes data pages back.

postgres removes WAL automatically once it's not needed anymore.

we can force a checkpoint at any time

Pros of checkpoint:

1)Reduce WAL File Size:

WAL keeps growing as changes are logged. A checkpoint writes these changes to the main database and clears older WAL entries, preventing excessive growth.

2)Improve Recovery Speed:

If a crash happens, recovery starts from the last checkpoint instead of the beginning of the WAL file. This makes recovery faster.

3)Ensure Data Consistency:

The checkpoint ensures that all committed transactions are permanently stored in the main database file.

Part of pg_waldump

```
rmgr: Btree    len (rec/tot):  72/  72, tx:    2969, lsn: 0/085C82B8, prev 0/085C8260,
desc: INSERT_LEAF off: 48, blkref #0: rel 1663/34405/2673 blk 11
rmgr: Btree    len (rec/tot):  72/  72, tx:    2969, lsn: 0/085C8300, prev 0/085C82B8,
desc: INSERT_LEAF off: 122, blkref #0: rel 1663/34405/2674 blk 5
rmgr: Heap     len (rec/tot):  203/ 203, tx:    2969, lsn: 0/085C8348, prev 0/085C8300,
desc: INSERT off: 5, flags: 0x00, blkref #0: rel 1663/34405/1259 blk 7
rmgr: Btree    len (rec/tot):  64/  64, tx:    2969, lsn: 0/085C8418, prev 0/085C8348,
desc: INSERT_LEAF off: 166, blkref #0: rel 1663/34405/2662 blk 2
rmgr: Btree    len (rec/tot):  80/  80, tx:    2969, lsn: 0/085C8458, prev 0/085C8418,
desc: INSERT_LEAF off: 46, blkref #0: rel 1663/34405/2663 blk 1
rmgr: Btree    len (rec/tot):  64/  64, tx:    2969, lsn: 0/085C84A8, prev 0/085C8458,
desc: INSERT_LEAF off: 330, blkref #0: rel 1663/34405/3455 blk 1
```

`pg_waldump -p path 00000001000000000000000008`

Needs to be run on terminal, not within PSQL.

hot update is an update on same page without hot we would shift to different page.

Described Anomalies and Isolation Levels

correct table is on slide 21

Isolation Levels	SQL Standard Anomalies				Serialization Anomalies	
	Dirty Write	Dirty Read	Non-Repeatable Read	Phantom Read	Lost Update	Write Skew
Read Uncommitted prevents	Yes	No	No	No	No	No
Read Committed prevents	Yes	Yes	No	No	No	No
Repeatable Read prevents	Yes	Yes	Yes	No	Depends on Implementation	No
Serializable prevents	Yes	Yes	Yes	Yes	Yes	Yes

Dirty Write?

A dirty write happens when one transaction overwrites another transaction's uncommitted changes.

```
/* Session 1 */
lesson=# begin;
BEGIN
lesson=# select t_payment from teacher where t_id = 1;
t_payment
-----
      23
(1 row)
```

```
/* Session 2 */
lesson=# begin;
BEGIN
lesson=# update teacher set t_payment = t_payment + 3 where t_id = 1;
UPDATE 1
lesson=# select t_payment from teacher where t_id = 1;
t_payment
-----
      26
(1 row)
```

```
lesson=# update teacher set t_payment=0 where t_id = 1;
```

T1 waits until T2 is committed or rolled back. After that this update can happen. So we do not have dirty write

T1 does not updates as long as T2 has not committed. → no dirty write

No Dirty Write

```
/* client session 1 – Agency pays teacher 1*/  
Begin;  
select t_payment from teacher where t_id = 1;  
/* some application code - calculating how much money teacher gets and doing the money  
transfer */  
select pg_sleep(20);  
update teacher set t_payment=0 where t_id = 1;  
commit;  
  
/* Client Session 2 – teacher reports back hours */  
Begin;  
update teacher set t_payment = t_payment + 3 where t_id = 1;  
SELECT t_payment FROM teacher WHERE t_id = 1;  
select pg_sleep(20);  
commit;
```

T1 firstly selects t_payment of teacher which let's say returns 6 and then sleeps for 20 seconds. while sleeping T2 begins it increments t_payment by 3 and then selects it and returns 6+3=9. then it sleeps 20 seconds. while sleeping T1 wakes up and tries to update t_payment, but since T2 has not committed or rolled back it won't update. then T2 wakes up and commits. Then T1 can see that T2 committed so it can update now and commit. So again dirty write is prevented.

Postgres Dirty Read?

A dirty read happens when a transaction reads data that has been modified but not yet committed by another transaction.

```
/* Session 1 */
```

```
lesson=# begin;
```

```
BEGIN
```

```
lesson=# select t_payment from teacher where t_id = 1;
```

```
t_payment
```

```
-----
```

```
0
```

```
(1 row)
```

```
/* Session 2 */
```

```
lesson=# begin;
```

```
BEGIN
```

```
lesson=# update teacher set t_payment = t_payment + 3 where t_id = 1;
```

```
UPDATE 1
```

```
lesson=# select t_payment from teacher where t_id = 1;
```

```
t_payment
```

```
-----
```

```
3
```

```
(1 row)
```

this is not committed so T1 does not read 9 because 9 could be rolled back

```
lesson=# select t_payment from teacher where t_id = 1;
```

```
t_payment
```

```
-----
```

```
0
```

```
(1 row)
```

If T2 commits: what will T1 return: 0 or 3?


```
/* Session 1 */
lesson=# begin isolation level repeatable read;
BEGIN
lesson=*# select t_payment from teacher where t_id = 1;
t_payment
-----
      0
(1 row)
```

```
/* Session 2 */
lesson=# begin isolation level repeatable read;
BEGIN
lesson=*# update teacher set t_payment = t_payment + 3 where t_id = 1;
UPDATE 1
lesson=*# select t_payment from teacher where t_id = 1;
t_payment
-----
      3
(1 row)
```

T2 commits. What value does T1 return for the same select query?

```
lesson=*# select t_payment from teacher where t_id = 1;
```

Isolation level repeatable read ensures that a transaction always sees the same data if it reads the same row multiple times even if other transactions modify it. So even if T2 commits T1 will still show 0.

SQL Standard Concurrency Problems

Dirty writes

A transaction overwrites data that another transaction has written, but not yet committed. All ACID implementations prevent dirty writes.

Dirty reads

A transaction reads another transaction's writes before they have been committed.

Non-Repeatable Reads:


A transaction sees different states of the database (different values for one object) at different points in time.

Phantom reads

A transaction reads objects that match some search condition. Another transaction makes a write (insert / delete) that affects the results of that search.

Phantom Read

ChatGPT prompt: Which postgres isolation level prevents phantom read?

ChatGPT answer: " In PostgreSQL, the only isolation level that prevents phantom reads is:  SERIALIZABLE

Why REPEATABLE READ Does NOT Prevent Phantom Reads? In REPEATABLE READ, PostgreSQL ensures that if a transaction reads the same row twice, the values remain unchanged. However, new rows matching the query condition can still be inserted by other transactions, leading to phantom reads."

Phantom Read

```
/* Phantom Read *: T1 starts before T2 but ends after T2 */
```

```
/* T1*/
```

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
SELECT COUNT(*) FROM teacher;
```

```
select pg_sleep(20);
```

```
SELECT COUNT(*) FROM teacher;
```

```
commit;
```

after T2 commits T1 still returns 20 since we have repeatable level so phantom read is prevented.

```
/* T2 */
```

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
INSERT INTO public.teacher(
```

```
    t_id, t_name, t_mail, t_postalcode, t_dob, t_gender, t_education)
```

```
    VALUES (21, 'Hall', 'hall@great.xx', 4006, '1950-03-08', 'm', 'Bachelor');
```

```
COMMIT;
```

Implementation difference between repeatable read and non-phantom read:
repeatable read requires an object lock (row lock), non-phantom read a predicate lock because one needs lock rows that are not yet there – if implemented with a locking technology.

with repeatable read you can lock what's there with phantom read you have to lock rows that are not even there. So we need predicate locks meaning we lock whatever falls under a search condition whether it's already there or it's coming in the future, does not matter

Anomalies and Isolation Levels PostgreSQL

Isolation Levels	SQL Standard Anomalies				Serialization Anomalies	
	Dirty Write	Dirty Read	Non-Repeatable Read	Phantom Read	Lost Update	Write Skew
Read Committed prevents	Yes	Yes	No	No	No	No
Repeatable Read prevents	Yes	Yes	Yes	Yes	Yes	No
Serializable prevents	Yes	Yes	Yes	Yes	Yes	Yes

Isolation Levels in Different Databases

PostgreSQL
Isolation Levels:

READ COMMITTED
REPEATABLE READ
SERIALIZABLE

MS SQL
Isolation Levels:

READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ
SNAPSHOT
SERIALIZABLE

- What isolation levels does the database / storage engine support?
- How are they implemented?
- What is the default? [in postgres: read committed](#)

Isolation Levels in Different Databases

INNODB (MariaDB)
isolation levels:

READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ
SERIALIZABLE

Oracle isolation levels:

READ COMMITTED

SERIALIZABLE

- What isolation levels does the database / storage engine support?
- How are they implemented?
- What is the default?

Lost Update Anomaly

What is a Lost Update?

- Two transactions concurrently perform a read-modify-write cycle.
- One transaction updates data (one or more rows) and commits.
- The second transaction updates the same data and overwrites the committed update without noticing / respecting the update of the first transaction.
- → Update of the first committed transaction is lost.

Preventing Lost Updates

Lost updates: What does the database do to prevent / what can the application do?

- Application has to set appropriate isolation level for transactions
- Which isolation level prevents Lost Updates?
 - PostgreSQL: [repeatable read](#)
 - Oracle: serializable,
 - MS SQL Server – Repeatable Read / snapshot isolation.
 - MySQL/ InnoDB: [serializable](#)
- What if the applications wants to prevent serialization errors?

[in postgres repeatable read prevents lost updates but it means that transaction may roll back so if we want our application to run through then we can force this by select for update.](#)

Preventing Lost Updates

- The application needs to prevent lost updates and set an exclusive lock on the object. Most relational databases provide a command to do this.

```
BEGIN TRANSACTION;
SELECT * FROM student WHERE name = 'Rose' FOR UPDATE;
UPDATE student SET counter =counter+5;
COMMIT;
```

This treats select like read and locks it,
updates it and then commits

select ... for update locks all rows that the query returns.

- Select for update
may reduce serialization errors and rollbacks. However, it may have negative effects on performance.

all the other transactions have to wait for the row to unlock

Preventing Lost Updates

- Some databases do not use the read-modify-write cycle but atomic operations.

Example: Mongodb offers atomic writes on the level of a single document.

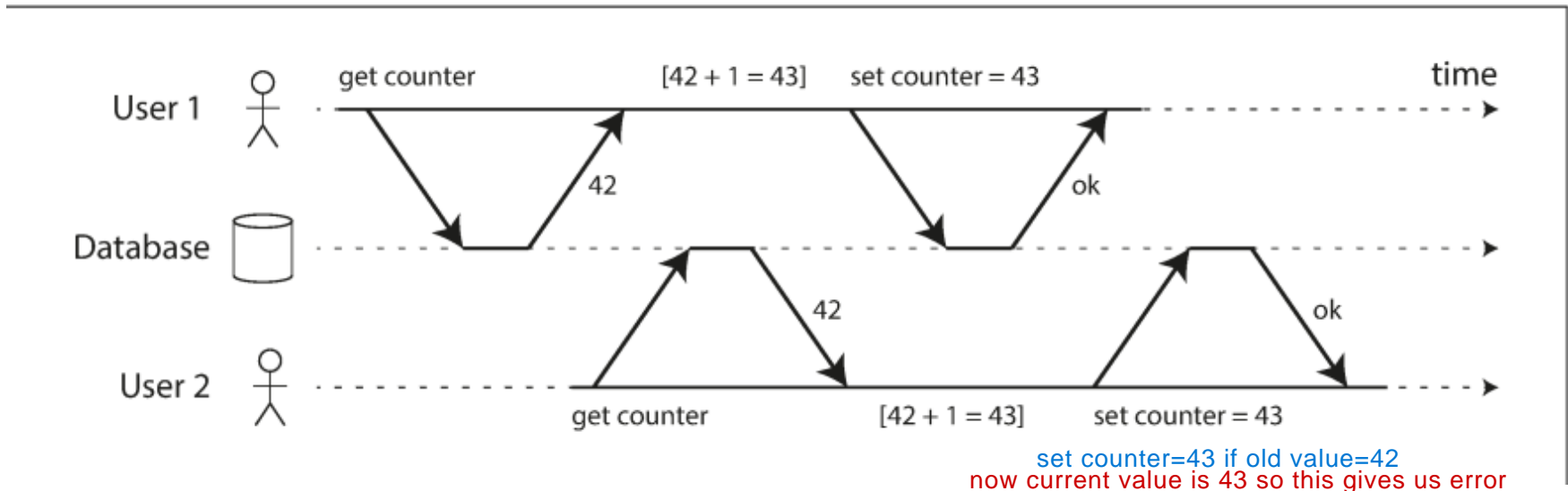
- `db.collection.updateOne()`
is an atomic write operation that can update multiple fields in the document including fields in all embedded documents
- `findAndModify` [similar to select...for update](#)
is also an atomic operation

Preventing Lost Updates

- Some distributed databases (that do not provide transaction support) may provide a compare-and-set feature for single-object writes: The old value is kept. Before an update is written back, the database checks whether the object has still the old value (=no other write has happened in the meantime).
- Some replicated databases only offer LWW.

Compare-and-Set

set counter=43 if old value=42



Some databases, like Cassandra, for example, do NOT support ACID transaction concurrency protocols to solve lost updates issues. But they offer at least atomic CAS (Compare-and-Set) operations to prevent Lost Updates.

Explain briefly what CAS means and how it works. Apply the CAS method to the lost update problem in the figure. Write down the (pseudo-code) CAS operation(s) that prevent the lost update.

Write Skew Anomaly

What is the difference between a Lost Update and a Write Skew Anomaly?

Example:

In order to operate properly, an airport needs a minimum of 100 security personnel on duty any time the airport is open. If this invariant cannot be kept, parts of the airport have to be shut down.

An application controls this constraint: Security personnel is only allowed to sign out off duty if minimum number of 100 personnel on duty is not violated.

One day, Alice and Bob are on the duty list but both want to sign out – for whatever reason. They have to check whether they are allowed to sign out.

Write Skew anomaly

1	T1	T2	Result
2	Begin		
3	n_on_duty = select count(*) from personnel where on_duty=TRUE	Begin	T1 returns 101 personnel on duty
4		n_on_duty = select count(*) from personnel where on_duty=TRUE	T2 returns 101 personnel on duty
5	if (n_on_duty > 100) (update personnel set on_duty = False where name= "Alice")		
6	commit;		100 personnel are on duty
7		if (n_on_duty > 100) (update personnel set on_duty = False where name= "Bob")	
8		Commit;	99 personnel are on duty

repeatable read does not work here because we are working on different rows!

postgres handle this with serialization level

Write Skew Anomaly

What is the difference between a Lost Update and a Write Skew Anomaly?

In write skew:

- write operation depends on a read return
- write operations is on different rows

unlike lost update here we don't update same data

Write skew can only be prevented if

- transactions run in serial execution mode, one after the other.
- transactions run in serializable mode equivalent to serial execution mode.

Transactions and Concurrency Control

1. Transactions and their ACID properties
 1. A
 2. C
 3. I
 4. D
2. Concurrency anomalies and isolation levels
3. Serializability

Concurrency Control Protocols

Pessimistic: Lock Based Protocols

Locking Protocols based on 2PL

- introduced about 50 years ago (1976)
- for about 30 years the only CC protocol used in commercial DBMS to enforce serializability.
- Widespread, benchmark for other protocols
- Today, MS SQL Server uses 2PL (default), MySQL / MariaDB partly use 2PL

Optimistic: Snapshot Isolation

MVCC (snapshot) protocols

- introduced about 30 years ago
- Today, used by most commercial DBMS
- PostgreSQL, Oracle, MySQL / MariaDB (partly)
- MVCC also uses locks – but differently

Lock-Based CC Protocols

A transaction acquires locks to lock the objects it wants to access. Other transactions have to wait until the objects are unlocked – depending on the lock and on the intended operation, read or write.

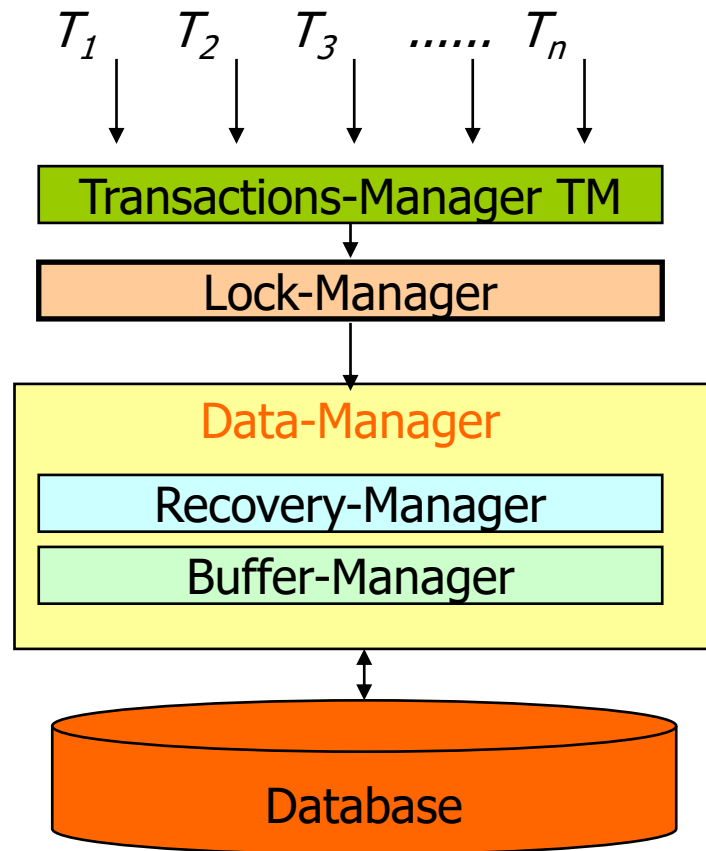
Two lock modes

Compatibility matrix

S (shared, read lock):

X (exclusive, write lock):

	NL	S	X
S	✓	✓	–
X	✓	–	–



The lock manager maintains lock tables with

- locked objects,
- transactions that have locked these objects, and
- the details of the lock mode.

The lock manager also keeps waiting lists (intention locks) with the transactions waiting to lock an object.