# Indexes and Transactions

1DL301 Database Design I

INTENDED LEARNING OUTCOMES

▶ To understand the purpose of indexes and how they work.

▶ To be able to determine when an index is needed and to be able to create it.

▶ To understand what transactions are, why and how they are used.

▶ To understand the ACID properties of transactions.

▶ To be able to use transactions in practice.

▶ To understand different isolation levels and related reading phenomena.
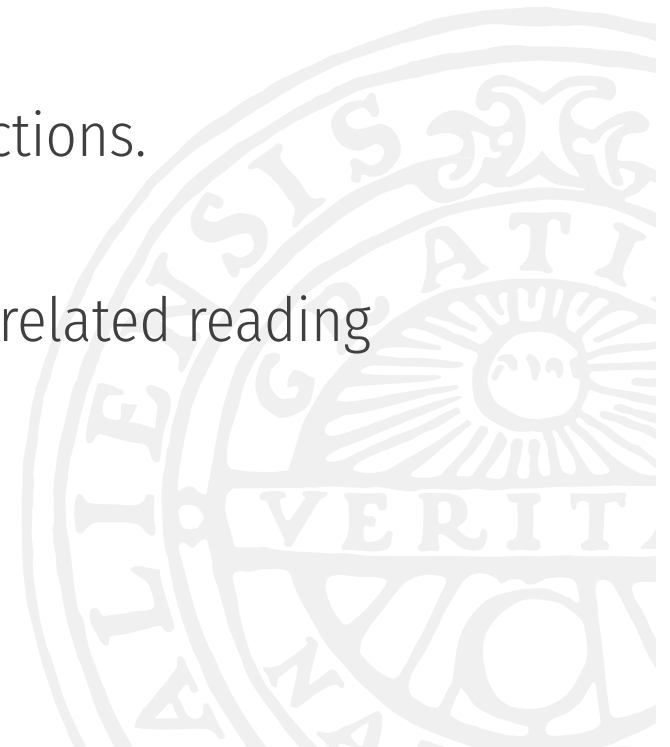
```
CREATE TABLE person(
  id int PRIMARY KEY,
  fname varchar(32) NOT NULL,
  lname varchar(32) NOT NULL,
  phone varchar(32)
);
```

MySQL database, 10 000 000 people. (Each row takes in average 50 bytes.)

▶ How long does it take to find name and phone number for person with ID 4,857,845?

▶ How long does it take to find the phone number for all people named Kristin Elvik?

```
SELECT *
FROM person
WHERE id=4857845;
+---------+--------+----------+------------+
| id      | fname  | lname    | phone      |
+---------+--------+----------+------------+
| 4857845 | Børre  | Heggheim | 9913139357 |
+---------+--------+----------+------------+
1 row in set (0.00 sec)


SELECT *
FROM person
WHERE fname='Kristin' AND lname='Elvik';
+---------+--------+-------+------------+
| id      | fname  | lname | phone      |
+---------+--------+-------+------------+
| 7350165 | Kristin | Elvik | NULL       |
| 9120140 | Kristin | Elvik | 3441093539 |
+---------+--------+-------+------------+
2 rows in set (3.74 sec)
```

3.74 sec is very slow. Why is it so slow and what can we do about it?

EXPLANATION

The slow query from the experiment performs a **full table scan**: each row in the table is read and checked if it meets the condition.

To see the execution plan (how the query is executed):

► EXPLAIN SELECT ... in MySQL

► EXPLAIN QUERY PLAN SELECT ... in SQLite

The execution plan of the slow query from the experiment:

```
EXPLAIN
SELECT *
FROM person
WHERE fname='Kristin' AND lname='Elvik';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	person	NULL	ALL	NULL	NULL	NULL	NULL	8815388	1.00	Using where

INDEX

Index – a persistent data structure which can be used to quickly locate rows in a table based on the values of one or several given attributes.

Similar idea to an index in a book – keywords are ordered alphabetically and for each keyword there is a list of pages.

Cost of an index:

▶ extra storage space (usually not a problem)

▶ index creation (might take time, but it is done only once)

▶ index maintenance – index must be updated when the data in the table changes (might be a problem)

CREATING AND DROPPING INDEXES

```
CREATE INDEX index_name ON table(column, column, ...);
CREATE UNIQUE INDEX index_name ON table(column, column, ...);
```

A unique index does not allow duplicate values.

Indexes on the primary key attributes are created automatically. Some RDBMS (e.g. MySQL) create indexes on the foreign key attributes automatically as well.

Different SQL syntax for dropping an index in different RDBMS:

SQLite:

```
DROP INDEX table.index_name;
```

MySQL:

```
ALTER TABLE table DROP INDEX index_name;
```

For other systems, check the reference manual.

```
CREATE INDEX lname_fname ON person(lname, fname);

SELECT *
FROM person
WHERE fname='Kristin' AND lname='Elvik';

+---------+--------+-------+------------+
| id      | fname  | lname | phone      |
+---------+--------+-------+------------+
| 7350165 | Kristin | Elvik | NULL       |
| 9120140 | Kristin | Elvik | 3441093539 |
+---------+--------+-------+------------+
2 rows in set (0.00 sec)


EXPLAIN
SELECT *
FROM person
WHERE fname='Kristin' AND lname='Elvik';
```
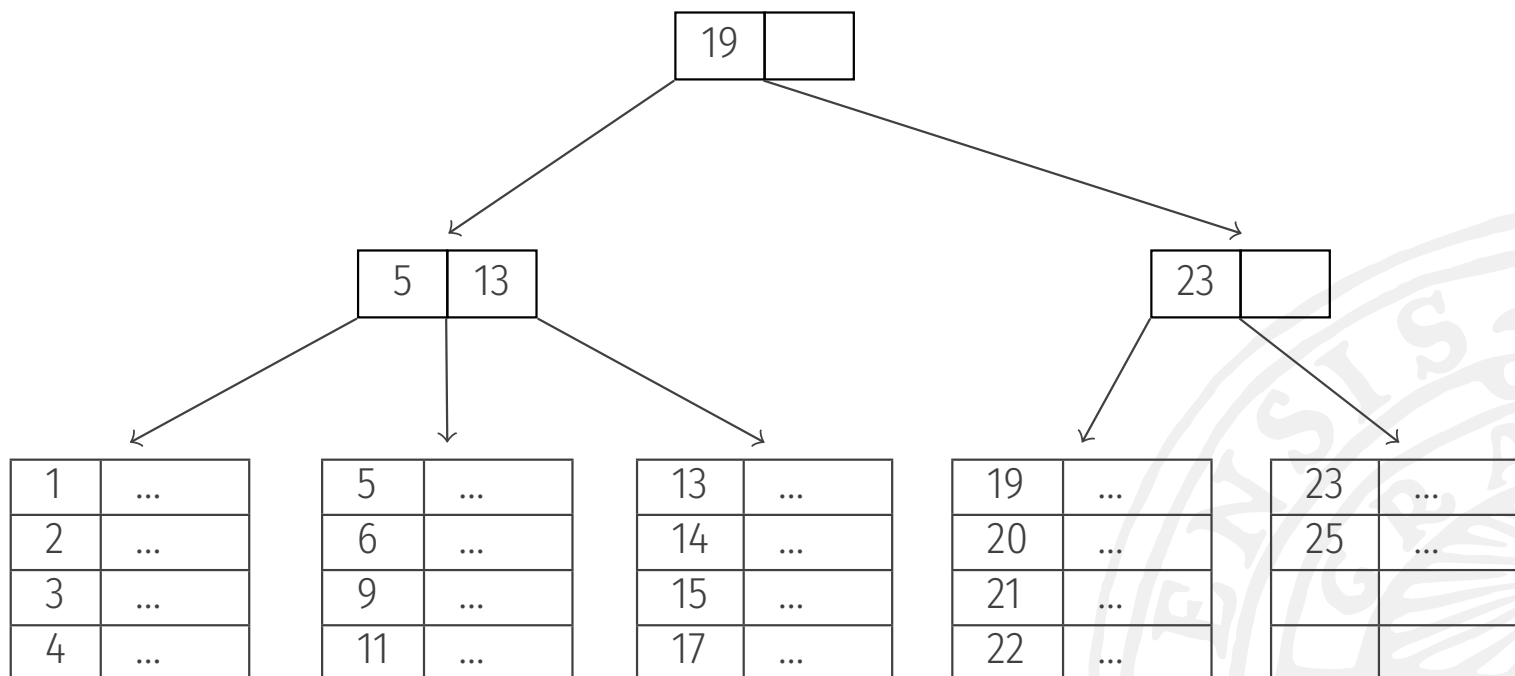
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	person	NULL	ref	lname_fname	lname_fname	196	const,const	2	100.00	NULL

Two main data structures being used for indexes:

▶ B and B+ trees – useful for conditions with $=, <, \leq, >, \geq$ and
  BETWEEN

▶ Hash tables – useful for conditions with $=$ only

CLUSTERED VS. NON-CLUSTERED INDEXES

Great explanation of clustered and non-clustered indexes:
https://youtube.com/watch?v=ITcOiLSfVJQ

Non-clustered index

▶ Physical order of the rows is not the same as the index order.

▶ Leaf level of the index points to where the data is stored.

Clustered index

▶ Physical order of the rows is the same as the index order.

▶ Used for indexing of primary keys.

▶ There might only be one clustered index per table.

▶ Fast retrieval of data when multiple rows match the condition.

DECIDING WHAT TO INDEX

▶ Which queries and insert/update/delete statements will be executed?

▶ How often?

▶ What are the time constraints?

▶ How many rows are in the relevant tables?

Tip: Add indexes which might be helpful and use

▶ `EXPLAIN` `SELECT` `...` in MySQL

▶ `EXPLAIN QUERY PLAN` `SELECT` `...` in SQLite

to see how the query is executed, what indexes are used and to check if the performance is satisfactory. Check execution time (with no cache). Drop indexes which are not used.

OTHER METHODS OF PERFORMANCE OPTIMIZATION

▶ Denormalization
$E(\underline{A_1}, A_2), F(\underline{B_1}, B_2), EF(\underline{A_1}, \underline{B_1}) \rightarrow EF(\underline{A_1}, \underline{B_1}, A_2, B_2)$

▶ Storing derived attributes

▶ Storing summary statistics
Example: Storing the number of employees for each department, storing the sum of hours_spent for each employee.
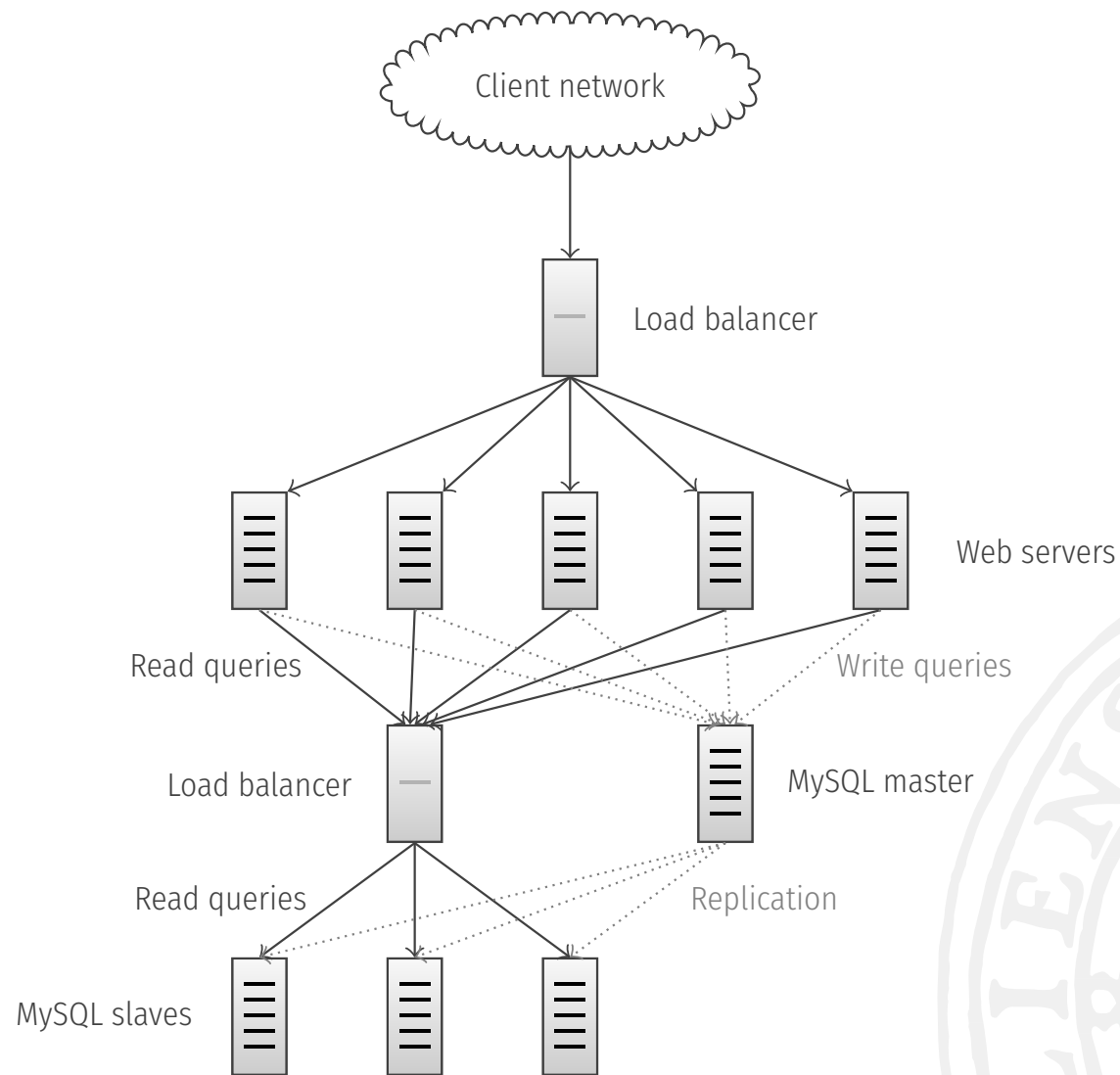
▶ Vertical partitioning
$E(\underline{A_1}, A_2, A_3) \rightarrow E_1(\underline{A_1}, A_2), E_2(\underline{A_1}, A_3)$. *$E_1$ and E_2 might be stored on separate physical location or database servers.*

▶ Horizontal partitioning (sharding)
Rows are divided into shards which might be stored on separate physical locations or database servers. E. g. logs are or records are partitioned based on the year when the log or record was created.

▶ Hardware tuning

Client network

Load balancer

Web servers

Read queries

Write queries

Load balancer

MySQL master

Read queries

Replication

MySQL slaves

From *High Performance MySQL: Optimization, Backups, and Replication* by Schwartz Harvey, Peter Zaitsev and Vadim Tkachenko

```
SELECT employee.*, department.title
FROM employee, department
WHERE employee.department_id=department.id
  AND department.title='Production A'
  AND employee.hour_salary < 200;
```

Which of the following indexes could NOT be useful?

1. Tree-based index on department.title
2. Hash-based index on department.title
3. Tree-based index on employee.hour_salary
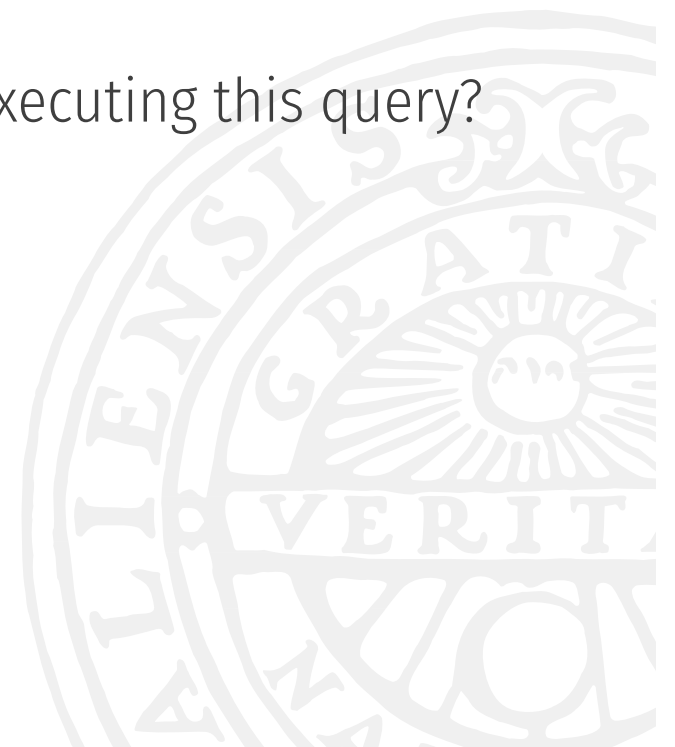4. Hash-based index on employee.hour_salary

```
CREATE INDEX emp_lname ON employee(last_name);

SELECT *
FROM employee
WHERE last_name LIKE '%g'
```

Is the index emp_lname going to be used when executing this query?

1. Yes
2. No

Consider a relation $R(\underline{A}, B, C, D)$ containing 10^7 records. A is the primary key, and B contains 10^5 distinct values. The following SQL prepared statements are executed very frequently:

```
UPDATE R SET D=? WHERE B=?
SELECT D FROM R WHERE C=?
```

Which indexes would you create?

1. One index on B and one on C
2. Index on B
3. One index on D and one on B
4. Index on (C, B)
5. Index on (B, D)

Consider a relation $R(\underline{A}, B, C, D)$ containing 10^7 records. B contains 10^5 distinct values. The following SQL prepared statement is executed very frequently:

```sql
SELECT C FROM R WHERE B=?
```

Which indexes would you create?

1. One index on B and one on C
2. Index on C
3. Index on B
4. Index on (B, C)
5. Index on (C, B)

EXERCISE: MONEY TRANSFER

Initial state of the Account table:

AccountNo	Name	Balance
12345678	Jan's checking account	3500
23456789	Jan's savings account	13000
...	...	...

Jan transfers 3000 SEK from his checking account to his savings account:

AccountNo	Name	Balance
12345678	Jan's checking account	500
23456789	Jan's savings account	16000
...	...	...

Write SQLs to execute this money transfer and discuss what can go wrong during their execution.

```
UPDATE Account
SET Balance=Balance-3000
WHERE AccountNo=12345678

UPDATE Account
SET Balance=Balance+3000
WHERE AccountNo=23456789
```

▶ The first SQL gets executed, followed by a software, hardware or network failure.

▶ Both SQLs get executed, but the hardware failure occurs before the changes are permanently stored on a storage device.

TRANSACTION

Transaction – a series of operations that need to be executed as a single unit of work and that transforms the database from one consistent (and valid) state to another.

Note that the integrity constraints might be violated during a transaction but not when it has finished.

Failure (or a user's decision to abort the transaction) must be treated as if the transaction never happened.

▶ **Commit** – making the "effects of the transaction" permanent (successful executing the transaction)

▶ **Rollback** – aborting the transaction (because of a failure or user's decision)

EXERCISE: CONCURRENCY

Jan has a subscription from Netflix which withdraws 100 SEK each month from Jan's checking account.

What can go wrong if this transfer happens to be executed at the same time as the transfer of money to Jan's savings account?

What are the possible balances at Jan's checking account after executing both transfers?
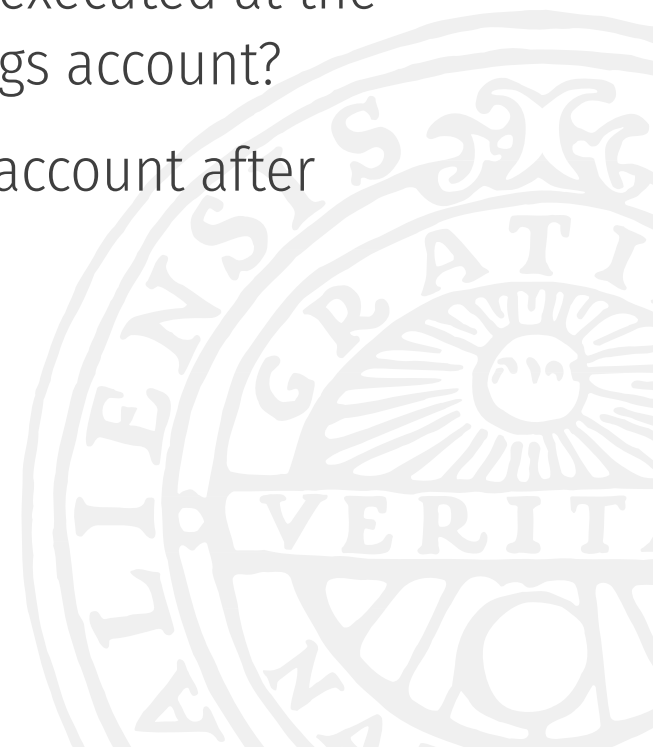
One of possible schedules:

Transfer 1	Transfer 2
READ *balance* of 12345678	
	READ *balance* of 12345678
$balance \leftarrow balance - 3000$	
	$balance \leftarrow balance - 100$
WRITE *balance*	
	WRITE *balance*
...	...

The account balance after both transfers will be 3400.

Other schedules might lead to the balance 500 and 400.

LOST UPDATE PROBLEM

T_1	T_2
READ a	
	READ a
UPDATE a	
	UPDATE a
WRITE a	
	WRITE a

The update of a in T_1 is "lost", a is overwritten by T_2 which read the state before updating and storing a in T_1.

CONCURRENCY

Databases are usually accessed (both queried and modified) by many clients at the same time.

To avoid problems with concurrent (parallel) execution (such as the lost update problem) a transaction schedule must be **serializable**:

Execution of concurrent transactions must be equivalent to <u>a</u> serial execution of the transactions.

Example: In our example, the execution must be equivalent either to transferring money to Jan's savings account first and then to Nexflix, or to Netflix first and then to Jan's savings account.

Note that in general different serial orders might lead to different results. If we care about the order we must run the transactions serially (rather than running them concurrently).

ACID

ACID – a set of required properties of transactions:

► **A**tomicity – all statements in a transaction are executed or none.

► **C**onsistency – all database constraints are satisfied after a transaction is executed.

► **I**solation – the result of executing concurrent transactions is the same as if the transactions were executed serially.

► **D**urability – once a transaction finishes, effects of the transaction are permanently stored in the database.

TRANSACTIONS IN SQL

▶ To start a transaction:

START TRANSACTION

or

BEGIN

▶ To commit the transaction (making the changes permanent):

COMMIT

▶ To roll back the transaction (canceling the changes):

ROLLBACK

EXAMPLE OF A TRANSACTION IN SQL

```
BEGIN;

UPDATE Account
SET Balance=Balance-3000
WHERE AccountNo=12345678;

UPDATE Account
SET Balance=Balance+3000
WHERE AccountNo=23456789;

COMMIT;
```

ISOLATION LEVELS

The ACID isolation property is often relaxed to reduce the overhead and increase the concurrency.

There are weaker isolation levels that allow some serializability violations in order to achieve higher performance.

Isolation level is set per transaction and it reflects what *read phenomena* might occur in the current transaction.

Read phenomena:

▶ Dirty reads

▶ Non-repeatable reads (also called inconsistent reads)

▶ Phantom reads

Reading data written by another transaction before it is committed:

T_1	T_2
| ```
SELECT stock FROM product
WHERE id=1;
-- returns 30
``` | |
| | ```
UPDATE product SET stock=29
WHERE id=1;
-- transaction is not finished yet
``` |
| ```
SELECT stock FROM product
WHERE id=1;
-- returns 29 (rather than 30)
``` | |
| | ```
ROLLBACK;
``` |

Retrieving the same row twice might return different data:

T_1	T_2
```SELECT stock FROM product WHERE id=1; -- returns 30```	
	```UPDATE product SET stock=29 WHERE id=1; COMMIT;```
```SELECT stock FROM product WHERE id=1; -- returns 29 (rather than 30)```	

New rows might appear in tables during the transaction:

T_1	T_2
| ```
SELECT stock FROM product
WHERE id BETWEEN 1 AND 100;
``` | |
| | ```
INSERT INTO product(id, stock)
VALUES (25, 100);
COMMIT;
``` |
| ```
SELECT * FROM product
WHERE id BETWEEN 1 AND 100;
-- includes <25, 100> as well
``` | |

ISOLATION LEVELS

	Dirty reads	Non-repeatable reads	Phantom reads
READ UNCOMMITTED	May occur	May occur	May occur
READ COMMITTED	—	May occur	May occur
REPEATABLE READ	—	—	May occur
SERIALIZABLE	—	—	—

▶ Default level for InnoDB in MySQL is REPEATABLE READ.

▶ SQL to change the isolation level (of the next transaction):

`SET TRANSACTION ISOLATION LEVEL level`

where *level* is:

▶ SERIALIZABLE
▶ REPEATABLE READ
▶ READ COMMITTED
▶ READ UNCOMMITTED

Which of the following is true if a transaction with only SELECT statements is executed at isolation level REPEATABLE READ?

1. `SELECT COUNT(*) FROM T` may return different results if executed multiple times inside the transaction.

2. `SELECT B FROM T WHERE A=1` may return different results if executed multiple times inside the transaction (A is the primary key).

3. A join that returns a non-empty table, when re-executed inside the transaction may return an empty result.

4. Nested queries may not be allowed, if another transaction has disabled them.

5. None of the other answers is true.

Which of the following is true if a transaction with only SELECT statements is executed at isolation level SERIALIZABLE?

1. `SELECT COUNT(*) FROM T` may return different results if executed multiple times inside the transaction.

2. `SELECT B FROM T WHERE A=1` may return different results if executed multiple times inside the transaction (A is the primary key).

3. A join that returns a non-empty table, when re-executed inside the transaction may return an empty result.

4. Nested queries may not be allowed, if another transaction has disabled them.

5. None of the other answers is true.

R(A) contains two rows: {(1), (2)}. The following transactions run concurrently:

T1: `UPDATE R SET A=2*A`

T2: `SELECT avg(A) FROM R`

If transaction T2 executes using the *read uncommitted* level, what are the possible values it returns?

1. 1.5, 2, 2.5, 3
2. 1.5, 2, 3
3. 1.5, 2.5, 3
4. 1.5, 3

Question by Jennifer Widom

$R(A)$ and $S(B)$ both contain two rows: {(1), (2)}. The following transactions run concurrently:

T1: `UPDATE R SET A=2*A; UPDATE S SET B=2*B`
T2: `SELECT avg(A) FROM R; SELECT avg(B) FROM S`

If transaction T2 executes using the *read committed* level, is it possible for T2 to return two different values?

1. Yes
2. No

Question by Jennifer Widom

QUIZ SOLUTION

Quiz 1: (4)

Quiz 2: (2)

Quiz 3: (1) is the best answer

Quiz 4: (3 & 4),

      (4) is the best answer, but needs deeper indexing knowledge

Quiz 5: (1 & 2)

Quiz 6: (5)

Quiz 7: (1)

Quiz 8: (1)

Exercise Uploaded

Ex-6-Transactions.pdf
  - George has Video Lectures:
    - Lect-11-zoom_0.mp4
    - Lect-12-zoom.mp4

ACKNOWLEDGEMENT

The slide is derived from
Jan Kudlicka's DB1-Fall19 class.