TRANSACTIONS AND DATABASE INTEGRITY

Module 3 : Database Operation

Test questions 1

en:

1. Definition and purpose of transactions.

2. Transaction properties.

3. Describe transaction concurrency issues:
   - loss update
   - dirty reading
   - inconsistent analysis

ru:

1. Определение и назначение транзакций.

2. Свойства транзакций.

3. Опишите проблемы параллельной работы транзакций:
   - потеря результатов обновления
   - чтение "грязных" данных
   - несовместимый анализ

Contents:

1. The concept of transaction
2. Transaction properties and commands
3. Transaction mix and launch schedule
4. Transaction Concurrency Issues
5. Methods for Solving Transaction Concurrency Issues
6. Transactions and data recovery

1. The concept of transaction

- **Transaction** is an indivisible sequence of data manipulation operations in terms of the impact on the database.

A logical unit of work:

For the user, the transaction is performed on the principle of "**all or nothing**": :

- or the whole transaction is **executed** and transfers the database from **one integral state** to **new integral state**,
- or, if one of the transaction actions is **not feasible**, or any system malfunction has occurred, the database **returns** to its **original state**, which was before the start of the transaction (the transaction is **rolled back**).

- Inside the transaction, integrity may be compromised.

Roll back:

Transactions are units of data recovery after failures - during recovery, the system **eliminates traces of transactions** that could not be completed normally as a result of a software or hardware failure.

Concurrency:

In **multiuser systems**, in addition, transactions serve to ensure the isolated work of users - users who simultaneously work with the same database, it seems that they work as if in a single-user system and do not interfere with each other.

Transactions provide system **Stability** and **Predictability**

2. Transaction properties and commands

ACID Properties:

(A) Atomicity . A transaction is done as an atomic operation - either the entire transaction is performed, or it is not being fully executed.

(C) Consistency . A transaction transfers the database from one consistent (integral) state to another consistent (integral) state.

(I) Isolation. A transactions of different users should not interfere with each other (for example, as if they were executed strictly by one by one).

(D) Durability. If the transaction is completed, then the results of its work should be saved in the database, even if the system crashes at the next moment.

Base integrity violation example

Inserting a new employee into the table does not *can be performed in one operation.* When you insert a new employee, you need to increase the value of the field at the same time DeptQty:

Step 1. Insert an employee into the table
PERSON: INSERT INTO PERSON (6, 'Milov', 2)

Step 2. Increase the value of the field DeptQty:
UPDATE DEPART SET DeptQty = DeptQty + 1 WHERE DeptId = 2

If, after performing the first operation and before performing the second, the system crashes, then only the first operation will actually be performed, and the database will remain in a non-integral state.

Depart

DeptId	DeptName	DeptQty
1	Programming department	2
2	Security department	1

Person

PersId	PersName	DeptId
1	Ivanov	1
2	Petrov	2
3	Sidorov	1

Transactions and SQL

The transaction starts automatically from the moment the user joins the database and continues until one of the following events occurs:

- The command **COMMIT** (commit transaction) was issued.
- The **ROLLBACK** command was given (roll back the transaction).
- A user disconnected from the DBMS.
- There was a failure of the system.

The **COMMIT** command completes the current transaction and automatically starts a new transaction. It is guaranteed that the results of the completed transaction are recorded, i.e. stored in the database.

The **ROLLBACK** command rolls back all changes made by the current transaction, i.e. canceled *as if they were not at all*. This automatically starts a new transaction.

When the user is disconnected from the database, transactions are automatically fixed.

- The **BEGIN** command marks the starting point of an explicit local transaction.
- The **SAVEPOINT** command for the current transaction sets a savepoint with the specified name

3. Transaction mix and launch schedule

A transaction is considered as a sequence of elementary atomic operations. The atomicity of a single elementary operation is that the DBMS ensures that, from the point of view of the user, two conditions are met:

- This operation will be performed completely or not at all (**atomicity** - all or nothing).
- During this operation, no other operations of other transactions are performed (**isolation** is a strict sequence of elementary operations).

In reality, the elementary operations of various transactions can be performed in random order. For example, there are several concurrent transactions consisting of a sequence of elementary operations.

$$T = (T_1, T_2, T_3, \ldots, T_n) \quad Q = (Q_1, Q_2, Q_3, \ldots, Q_m) \quad S = (S_1, S_2, S_3, \ldots, S_l)$$

Transaction mix

$$(T_1, Q_1, T_2, S_1, T_3, S_2, S_3, Q_2, \ldots)$$

Launch schedule

Definition 1. A set of several transactions whose elementary operations alternate with each other is called a ***t*ransaction mix** .

Definition 2. The sequence in which the elementary operations of a given set of transactions are performed is called the **launch schedule** for the transaction mix.

Note. For a given set of transactions, there can be several (generally speaking, quite a lot) different launch schedules.

- Ensuring user isolation is reduced to choosing **an appropriate transaction launch schedule**.

Definition *3*. A schedule for launching a set of transactions is called a **serial schedule** if transactions are performed strictly in turn, that is, elementary transactions are not alternated with each other.

Definition 4. If a schedule for starting a set of transactions contains alternating elementary transactions of transactions, then this schedule is called **nonserial schedule** (interleaving)**.**

4. Transaction Concurrency Issues

How can transactions of different users interfere with each other?

There are three main problems of concurrency:

- lost update
- uncommitted dependency (dirty reading)
- inconsistent analysis

Let's take a closer look at these issues.

Designations

- Consider two transactions A and B, starting in accordance with some schedules.
- Let transactions work with some database objects, such as table rows.
- The operation of **reading** the row **P** will be denoted by **P = P0**, where **P0** is the value read.
- The operation of **writing** the value of **P1** to the row **P** will be denoted by **P1 -> P**.

Lost update problem

- Two transactions take turns writing some data on the same row and committing the changes.

Transaction A	Time	Transaction B
Read $P = P_0$	t_1	---
---	t_2	Read $P = P_0$
Write $P_1 \rightarrow P$	t_3	---
---	t_4	Write $P_2 \rightarrow P$
Commit	t_5	---
---	t_6	Commit
Lost update result		

Result. After both transactions are completed, row P contains the value P2, obtained by the later transaction B. Transaction A knows nothing about the existence of transaction B and, naturally, expects row P to contain the value P1. Thus, transaction A lost the results of its work.

Uncommitted dependency

(dirty reading)

Transaction B modifies the data in the row. After that, transaction A reads the changed data and works with them. Transaction B rolls back and restores old data.

Transaction A	Time	Transaction B
---	t_1	Read $P = P_0$
---	t_2	Write $P_1 \rightarrow P$
Read $P = P_1$	t_3	---
Work with read data P_1	t_4	---
---	t_5	Roll back $P_0 \rightarrow P$
Commit	t_6	---
Work with dirty data		

What did transaction A work with?

Uncommitted dependency

(dirty reading)

Result. Transaction A in its work used data that is not in the database.

Moreover, transaction A used data that was not there and was not in the database!

Indeed, after the rollback of transaction B, the situation should be restored, as if transaction B had never been executed at all.

Thus, the results of transaction **A** are **incorrect**, because it worked with **data** that **was not in the database**.

The problem of incompatible analysis

- Unrepeatable reading.
- Fictitious elements (phantoms).
- Actually incompatible analysis.

Unrepeatable reading

Transaction A reads the same row twice. Between these readings, transaction B wedges in, which changes the values in the row.

Transaction A	Time	Transaction B
Read P = P_0	t_1	---
---	t_2	Read P = P_0
---	t_3	Write $P_1 \rightarrow P$
---	t_4	Commit
Second Read P = P_1	t_5	---
Commit	t_6	---
Unrepeatable reading		

Unrepeatable reading

Transaction A knows nothing about the existence of transaction B, and since it does not change the value in the row, it expects the value to be the same after a second read.

Result. Transaction A works with data that, from the point of view of transaction A, changes spontaneously.

Fictitious elements (phantoms)

Transaction A selects rows with the same conditions twice. Transaction B wedges between samples, adds a new row that satisfies the selection condition.

Transaction A	Time	Transaction B
Selection of rows satisfying condition a (n rows selected)	t_1	---
---	t_2	Insert a new row that satisfies condition a
---	t_3	Commit
Selection of rows satisfying condition a (n+1 rows selected)	t_4	---
Commit	t_5	---
Rows that didn't exist before appear		

Transaction **A** knows nothing about the existence of transaction **B**, and since it itself does not change anything in the database, it expects that the same rows will be selected after reselecting.

Result. Transaction **A** in two identical row samples received different results.

Actually incompatible analysis

The mixture contains two transactions - one long, the other short.

A long transaction performs some analysis throughout the table, for example, calculates the total amount of money on the bank accounts for the chief accountant. Let all accounts have the same amount, for example, $100. A short transaction at this point transfers $50 from one account to another, so the total amount for all accounts does not change.

Transaction A	Time	Transaction B
Read account P_1 = 100 and summing. Sum = 100	t_1	---
---	t_2	Withdraw money from P_3 account. P_3: 100 → 50
---	t_3	Putting money into P_1 account. P_1: 100 → 150
---	t_4	Commit
Read account P_2 = 100 and summing. Sum = 200	t_5	---
Read account P_3 = 50 and summing. Sum = 250	t_6	---
Commit	t_7	---
The total amount of $250 is incorrect – should be $300		

Actually incompatible analysis

Result. Although transaction B did everything right - the money was transferred without loss, but as a result, transaction A calculated the wrong total amount.

Since money transfer operations are usually continuous, in this situation it should be expected that the chief accountant will **never** know how much money is in the bank.

Competing transactions

- An analysis of the problems of concurrency shows that if no special measures are taken, then when working in a mixture, the property (I) of the transaction is insulated. Transactions really prevent each other from getting the right results.
- However, not all transactions interfere with each other. Transactions do not interfere with each other if they turn *to different data* or run in *different times.*
- ***Definition.*** Transactions are called **Competing**, if they intersect over time and turn to the same data.

Conflicts between transactions

- As a result of competition for data between transactions, there is a *Access conflicts* To the data:
- **W-W (Write - Write).** The first transaction changed the object and did not end there. The second transaction tries to change this object.
  Result. Lost update.
- **R-W (Read - Write).** The first transaction read the object and did not end. The second transaction tries to change this object.
  Result. Inconsistent analysis (unrepeatable reading).
- **W-R (Write - Read).** The first transaction changed the object and did not end there. The second transaction tries to read this object.
  Result. Dirty reading.
- **R-R (Read - Read) .**There are no conflicts, because reading data does not change.

Test questions$_1$

en:

1. Definition and purpose of transactions.
2. Transaction properties.
3. Describe transaction concurrency issues:
   - loss update
   - dirty reading
   - inconsistent analysis

ru:

1. Определение и назначение транзакций.
2. Свойства транзакций.
3. Опишите проблемы параллельной работы транзакций:
   - потеря результатов обновления
   - чтение "грязных" данных
   - несовместимый анализ

Test questions$_2$

en:

1. Describe ways to solve transaction concurrency issues using locks.

2. Describe the algorithm for recovering a database after a mild failure.

ru:

1. Опишите способы решеня проблем параллельной работы транзакций с использованием блокировок.

2. Опишите алгоритм восстановления базы данных после мягкого сбоя.

METHODS FOR SOLVING TRANSACTION CONCURRENCY ISSUES

How to resolve competition

Since transactions do not interfere with each other if they access **different data** or are executed at **different times**, there are two ways to allow competition between transactions arriving at arbitrary moments:

1. **"Slow down"** some incoming transactions as much as necessary to ensure the correct combination of transactions at each moment in time (that is, to ensure that **competing transactions** are executed at **different times**).

2. Provide competing transactions with **different data instances** (i.e. make sure that competing transactions work with **different versions of the data**).

The first method - "slowing down" transactions - is implemented using various types of **locks** or the **timestamp method**.

The second method - "providing different versions of the data" - is implemented using data from the **transaction log**.

Locks

There are two types of locks:

Exclusive locks (*X-locks*) - locks without mutual access **(write lock).**

Shared locks (***S-locks***) - mutual access locks (**read lock**).

If transaction A locks an object using **X-lock**, then any access to this object from other transactions is **rejected**.

If transaction A locks the object using **S-lock**, then:

- requests from other transactions for **X-lock** of this object is **rejected**,
- requests from other transactions for the **S-lock** of this object is **accepted**.

	Transaction B is trying to lock:	
Transaction A has locked:	**S-Lock**	**X-lock**
S-Lock	Yes	NO (Conflict R-W)
X-lock	NO (Conflict W-R)	NO (Conflict W-W)

Data access protocol

Before **reading** an object, a transaction must **impose** an **S-lock** on this object.

Before **updating** the object, the transaction must **impose** an **X-lock** on this object. **If** the transaction **has** already locked the object using **S-lock** (for reading), then before updating the object, the S-lock should be **replaced** with an **X-lock**.

If object **lock** by transaction **B** is **rejected** because the object is **already locked** by transaction **A**, then transaction **B** enters a **wait** state. Transaction **B** will be **waiting until** transaction **A unlocks** the object.

X-locks imposed by transaction **A** are **retained** until the **end** of transaction **A**.

Solving transaction concurrency issues

Loss update problem

Two transactions take turns writing some data on the same row and committing the changes.

Transaction A	Time	Transaction B
S-lock P - successful	t_1	---
Read P = P_0	t_2	---
---	t_3	S-lock P - successful
---	t_4	Read P = P_0
X-lock P - rejected	t_5	---
Waiting ...	t_6	X-lock P - rejected
Waiting ...	T_7	Waiting ...
Waiting ...		Waiting ...

Result. Both transactions are waiting for each other and cannot continue. There was a **deadlock** situation.

Uncommitted dependency problem (dirty reading)

Transaction B modifies the data in the row. After that, transaction A reads the changed data and works with them. Transaction B rolls back and restores old data.

Transaction A	Time	Transaction B
---	t_1	S-lock P - successful
---	t_2	Read P = P_0
---	t_3	X-lock P - successful
---	t_4	Write $P_1 \rightarrow P$
S-lock P - rejected	t_5	---
Waiting …	t_6	Rollback $P_0 \rightarrow P$ (Unlock P)
S-lock P - successful	T_7	---
Read P = P_0	T_8	---
Work with read data P_0	T_9	---
---	T_{10}	---
Commit	T_{11}	---
OK		

Result.
Problem resolved

Unrepeatable reading

Transaction A reads the same row twice. Between these readings, transaction B wedges in, which changes the values in the row.

Transaction A	Time	Transaction B
S-lock P – successful	t_1	---
Read P = P_0	t_2	---
---	t_3	X-lock P – rejected
---	t_4	Waiting …
Second Read P = P_0	t_5	Waiting …
Commit (Unlock P)	t_6	Waiting …
---	T_7	X-lock P – successful
---	T_8	Write $P_1 \rightarrow P$
---	T_9	Commit (Unlock P)
OK		

Result.
Problem resolved

The problem of incompatible analysis

Fictitious elements (phantoms)

Transaction A selects rows with the same conditions twice. Transaction B wedges between samples, adds a new row that satisfies the selection condition.

Transaction A	Time	Transaction B
S-lock rows satisfying condition a (n rows locked)	t_1	---
Selection of rows satisfying condition a (n rows selected)	t_2	---
---	t_3	Insert a new row that satisfies condition a
---	t_4	Commit
S-lock rows satisfying condition a (n+1 rows locked)	t_5	---
Selection of rows satisfying condition a (n+1 rows selected)	t_6	---
Commit	t_7	---
Rows that didn't exist before appear		

Result. Row level locking didn't solve the problem of fictitious elements

Actually incompatible analysis

The effect of the incompatible aalysis itself is also different from previous examples in that there are two transactions in the mix - one long, the other short.

A long transaction performs some analysis throughout the table, for example, calculates the total amount of money on the bank accounts for the chief accountant. Let all accounts have the same amount, for example, $100. A short transaction at this point transfers $50 from one account to another, so the total amount for all accounts does not change.

Actually incompatible analysis

Transaction A	Time	Transaction B
S-lock P_1 - successful	t_1	---
Read account P_1 = 100 and summing. Sum = 100	t_2	---
---	t_3	X-lock P_3 - successful
---	t_4	Withdraw money from P_3 account. P_3: 100 ⟶ 50
---	t_5	X-lock P_1 - rejected
---	t_6	Waiting ...
S-lock P_2 - successful	t_7	Waiting ...
Read account P_2 = 100 and summing. Sum = 200	t_8	Waiting ...
S-lock P_3 - rejected	t_9	Waiting ...
Waiting ...	t_{10}	Waiting ...

Result. Both transactions are waiting for each other and cannot continue. There was a **deadlock** situation.

Problem analysis

- Loss update problem - There was a deadlock situation.
- Uncommitted dependency problem (dirty reading) - Problem resolved.
- Unrepeatable reading problem - Problem resolved.
- The appearance of fictitious elements - Problem was not solved.
- The problem of incompatible analysis - There was a deadlock situation.

General view of the dead lock

Transaction A	Time	Transaction B
Lock object P_1 – successful	t_1	---
--	t_2	Lock object P_2 – successful
P_2 object lock conflicts with a lock imposed by transaction A	t_3	---
Waiting …	t_4	P_1 object lock conflicts with a lock imposed by transaction A
Waiting …	t_5	Waiting …
Waiting …		**Waiting …**

Because there is no normal way out of the deadlock situation, then such a situation needs to be recognized and eliminated. A method for resolving a deadlock situation is to **roll back** one of the transactions (**victim transaction**) so that other transactions continue their work. After resolving the deadlock, the transaction selected as the victim can be **repeated again**.

Two approaches for choosing a victim

1. The DBMS does not monitor the occurrence of deadlocks. Transactions themselves decide whether to be their victim.
2. The DBMS itself monitors the occurrence of a deadlock situation, it also decides which transaction will be the victim.

Resolving the remaining problems

The remaining problems, in particular phantoms, are solved by blocking an object of a larger size than the lines.

For example, locking at the **column** level, **multiple rows**, **tables**, **databases**. When blocking large database objects, there are **fewer opportunities** for parallel transactions..

When using locks of objects of different sizes, the problem of detecting **already imposed locks** arises. If transaction A is trying to lock the table, then you need to have information if there are already locks at the row level of this table that are incompatible with table locking.

To solve this problem, **the intentional locking protocol** is used, which is an extension of the data access protocol. The essence of this protocol is that before imposing a lock on an object (for example, on a row in a table), it is necessary to impose a **special intentional lock** (intent lock) on objects that include a locked object.

Two-phase transaction confirmation

In distributed systems, committing transactions may require the interaction of several processes on different machines, each of which stores some variables, files, databases. To achieve the indivisibility of transactions in distributed systems, a special protocol is used called **the two-phase transaction fixing protocol**. Although it is not the only protocol of its kind, it is most widely used.

- **In the first phase**, one of the processes acts as a **coordinator**. The coordinator starts the transaction by recording this in his logbook, then he sends to all subordinate processes that are also performing this transaction a message "**Prepare for commit**". When subordinate processes receive this message, they check to see if they are ready for committing, make an entry in their log and send the coordinator a response message "**Ready for commit**". After that, the subordinate processes remain in a ready state and wait for the commit command from the coordinator. If at least one of the subordinate processes has not responded, the coordinator rolls back the subordinate transactions, including those that are prepared for fixing.
- **The second phase** is that the coordinator sends a **Commit** command to all subordinate processes. By executing this command, the latter commit the changes and complete the subordinate transactions. As a result, simultaneous synchronous completion (successful or unsuccessful) of a distributed transaction is guaranteed.

TRANSACTIONS AND DATA RECOVERY

After the system fails, the subsequent **launch analyzes** the transactions that were performed before the transaction fails.

- Those transactions for which the **COMMIT** command **was given**, but whose work results were not recorded in the database, are executed again (**rolled**).
- Those transactions for which the **COMMIT** command **was not given** are **rolled back**.

Data durability

- The requirement of **data durability** (one of the properties of transactions) is that the data of **completed** transactions **must be stored** in the database, even if the system crashes at the next moment.
- The requirement of **atomicity** of transactions states that **incomplete** or rollback transactions **should not leave traces** in the database. This means that the data must be stored in the database with **redundancy**, which allows you to have information from which you **can restore** the state of the database at the time of the start of a failed transaction.
- This redundancy is usually provided by the **transaction log**. The transaction log **contains details of all data modification operations** in the database, in particular, the old and new values of the modified object, the system number of the transaction that modified the object and other information..

Types of failures

- **Individual transaction rollback.** It can be initiated either by the transaction itself using the ROLLBACK command, or by the system. The DBMS can initiate a transaction rollback in case of any error in the transaction operation (for example, **division by zero**) or if this transaction is selected as a **victim** when resolving the deadlock.
- **Mild system failure (software failure).** It is characterized by the loss of system RAM. In this case, all transactions that are performed at the time of the failure are affected, the contents of all database buffers are lost. Data stored on disk remains intact. A mild failure can occur, for example, as a result of a **power outage** or as a result of a **fatal processor failure**.
- **Hard system failure (hardware failure).** It is characterized by damage to external storage media. It can occur, for example, as a result of a **breakdown of the heads of disk drives**.

Transaction log

In all three cases, the basis of recovery is the **redundancy** of data provided by the **transaction log**.

Like database pages, data from the transaction log is not written directly to disk, but is **pre-buffered in RAM**. The system supports two types of buffers:

- database page buffers,
- transaction log buffers.

Logging

Database pages whose contents in the buffer (in RAM) are different from the contents on the disk are called **dirty pages**.

The system constantly maintains a list of dirty pages - **a dirty list**.

Writing dirty pages from the buffer to disk is called **pushing pages into external memory**.

The basic principle of a consistent policy for pushing the log buffer and database page buffers is that the record about the change of the database object must fall into the external memory of the **log before the changed** object is in the external memory of the **database**.

The corresponding logging (and buffering control) protocol is called **Write Ahead Log (WAL)** - "write first to the log", and consists in the fact that if you want to push the modified database object into external memory, you must first ensure that the log is pushed into external memory records of its change.

Save checkpoint

- Additional condition for pushing buffers:

Each successfully **completed** transaction must be actually **saved** in external memory. Whatever failure occurs, the system should be able to restore the state of the database containing the results of all transactions **committed** at the time of the failure.

- The third condition for pushing buffers is:

Limited volume of database buffers and transaction logs. The system accepts a checkpoint, which includes pushing the contents of the database buffers into an external memory and a special **physical record of the checkpoint**, which is a list of all transactions currently being performed.

- The minimum requirement guaranteeing the possibility of restoring the last consistent state of the database is **to push** all the database change records by this transaction when the transaction is **committed to the external memory**. At the same time, the last log entry made on behalf of this transaction is a special record about the end of this transaction

Individual transaction rollback

In order to be able to perform an individual rollback of a transaction in the transaction log, **all log records** from this transaction are **linked to the reverse list**.

The beginning of the list for non-completed transactions is a record of the last database change made by this transaction.

For completed transactions (individual rollbacks of which are no longer possible), the beginning of the list is a record of the end of the transaction, which is necessarily pushed into the external memory of the log.

The end of the list is always the first record of a database change made by this transaction. Each record has a unique transaction system number so that you can restore a direct list of records of database changes for this transaction.

Individual transaction rollback (algorithm)

1. A **list of records** made by a given transaction in the transaction log is **viewed** (from the last change to the first change).

2. The **next record** is **selected** from the list of this transaction.

3. The **opposite operation** is performed: instead of the INSERT operation, the corresponding DELETE operation is performed, instead of the DELETE operation, INSERT is performed, and instead of the direct UPDATE operation, the inverse UPDATE operation restores the previous state of the database object.

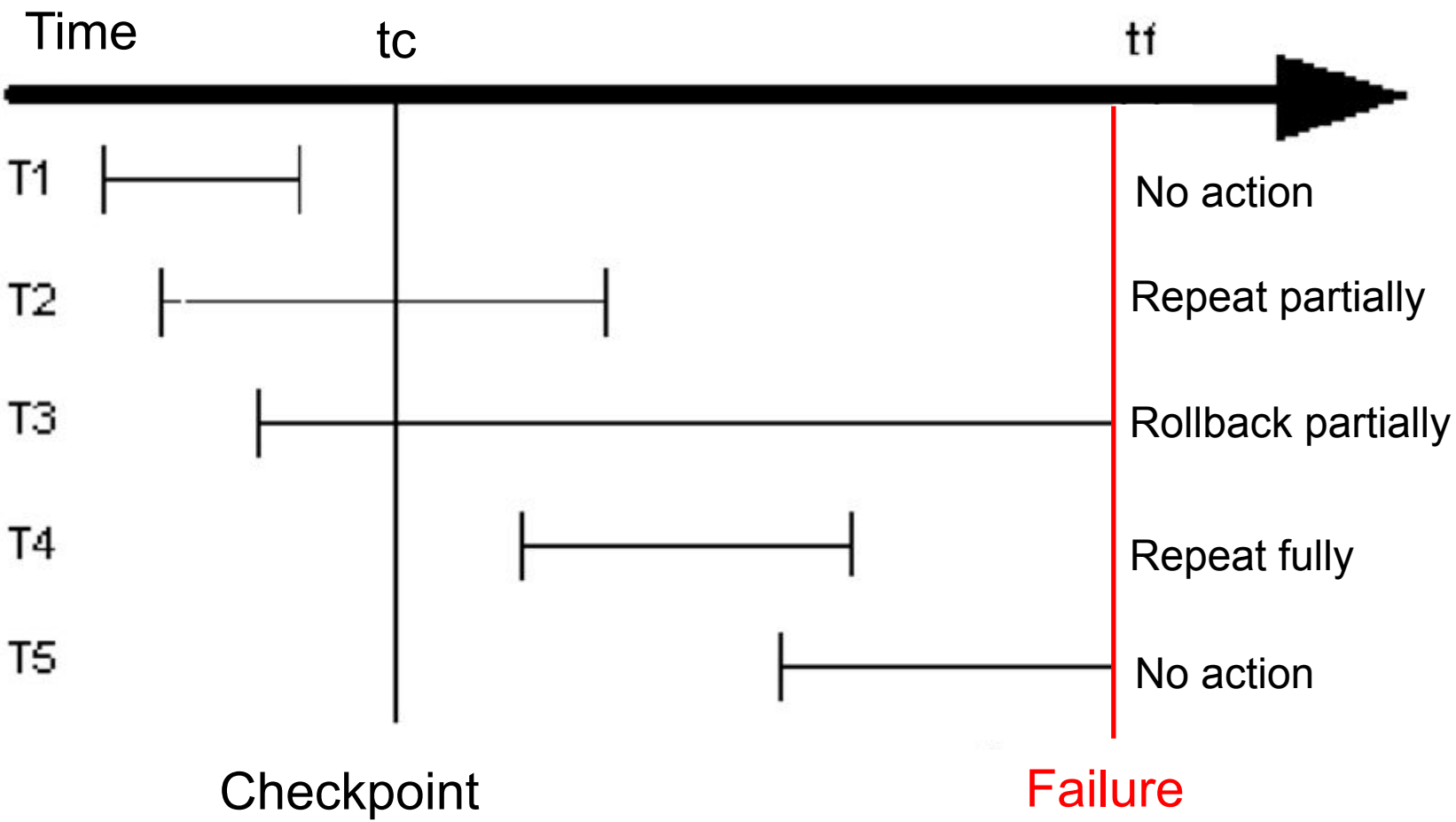
4. Any of these **reverse operations** are also **logged**. This must be done, because during the execution of an individual rollback, a mild failure may occur, during recovery after which it will be necessary to roll back a transaction for which an individual rollback has not been fully completed.

5. Upon successful completion of the rollback, a **record of the end** of the transaction is **logged**.

Recovering from a mild failure

- After a **mild failure**, not all physical database pages contain changed data, because **not all dirty database pages** were **pushed** to external memory.
- The last moment when the dirty pages were guaranteed to be pushed out is the moment of the **adoption** of the **last checkpoint**. There are 5 options for the state of transactions with respect to the time of the last checkpoint and the time of failure:

Recovering from a mild failure



Recovering from a mild failure

The last checkpoint was taken at time tc. A mild system failure occurred at time tf. Transactions T1-T5 are characterized by the following properties:

T1. The transaction completed **successfully** before the adoption of the checkpoint. All data of this transaction is stored in long-term memory - both log records and data pages changed by this transaction. Transaction T1 **does not require any recovery operations**.

T2 . The transaction **started before** the adoption of **checkpoint** and successfully **completed** after the checkpoint, but **before the failure**. The transaction log records related to this transaction are pushed to external memory. Data pages modified by this transaction are only partially pushed into external memory. For this transaction, it is necessary **to repeat again the operations that were performed after the adoption of the checkpoint**.

- **T3**. The transaction **started before** the adoption of the checkpoint and **was not completed** as a result of the **failure**. Such a transaction **must be rolled back**. The problem, however, is that some of the data pages modified by this transaction are already contained in the external memory - these are the pages that were updated before the adoption of the checkpoint. There are no traces of changes made after the checkpoint in the database. Transaction log entries made prior to the adoption of the checkpoint are pushed to external memory, those log records that were made after the checkpoint are not in the external memory of the log.

Recovering from a mild failure

T4. The transaction started **after the adoption of the checkpoint** and successfully **completed before the system failure**. The transaction log records related to this transaction are pushed into the external log memory. Changes to the database made by this transaction are completely absent in the external memory of the database. This **transaction must be repeated i**n its **entirety**.

T5 . The transaction **started after** the adoption of the **checkpoint** and **was not completed** as a result of the failure. There are no traces of this transaction either in the external memory of the transaction log, or in the external memory of the database. For such a transaction, **no action** needs to be taken, as if it did not exist at all.

System recovery after a mild failure is performed as part of the system reboot procedure. When the system is rebooted,

- transactions T2 and T4 - must be partially or completely repeated,
- transaction T3 is partially rolled back,
- no action is required for transactions T1 and T5.

Recovering from a hard system failure

- If a hard failure occurs, the database on the disk is **physically disrupted**. The basis of recovery in this case is the **transaction log** and an archive **copy of the database**. An archive copy of the database should be created periodically taking into account the speed of filling the transaction log.
- Recovery **begins** with backing up the database from the **archive copy**. **Then, a transaction log** is reviewed to identify all transactions that **completed successfully before the failure**. (Transactions ending with rollback before the failure can not be considered). After that, the transaction log in the forward direction repeats all successfully completed transactions. At the same time, there is no need to roll back transactions interrupted as a result of a failure, because the changes made by these transactions are not available after restoring the database from the backup.
- The worst case is when **both the database and the transaction log are physically destroyed**. In this case, the only thing that can be done is to restore the state of the database at the time of the **last backup**. In order to prevent this situation from occurring, the database and the transaction log are usually located on **physically different disks** managed by physically different controllers.

Test questions$_2$

en:

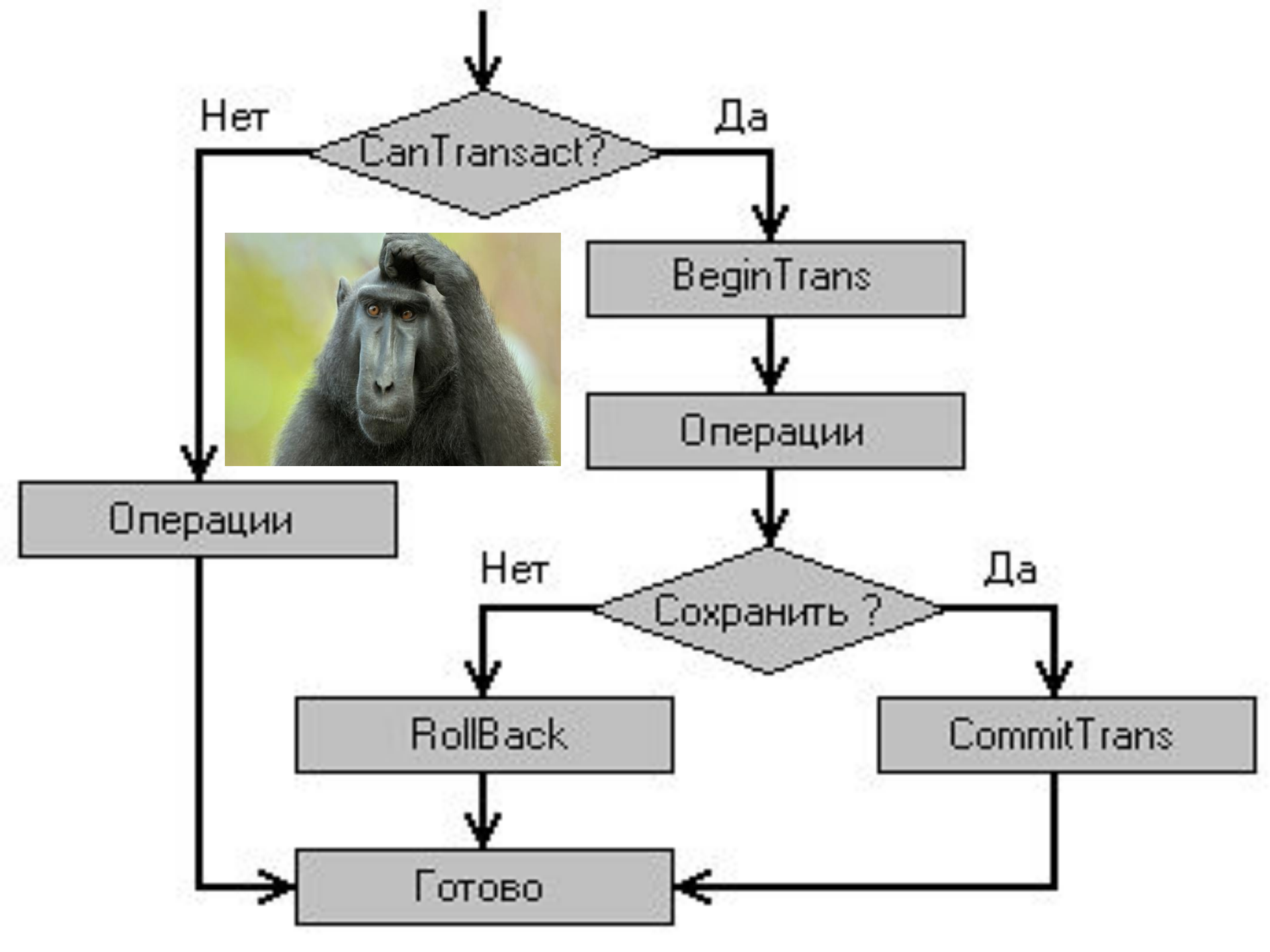
1. Describe ways to solve transaction concurrency issues using locks.

2. Describe the algorithm for recovering a database after a mild failure.

ru:

1. Опишите способы решеня проблем параллельной работы транзакций с использованием блокировок.

2. Опишите алгоритм восстановления базы данных после мягкого сбоя.


Нет
CanTransact?
Да
BeginTrans
Операции
Операции
Нет
Сохранить ?
Да
RollBack
CommitTrans
Готово