Database and Information Systems

Michal Krátký & Radim Bača

Department of Computer Science
Faculty of Electrical Engineering and Computer Science
VŠB – Technical University of Ostrava

2020/2021

Content

1. Concurrency control problems
   - Lost update
   - Uncommitted dependency
   - Inconsistent analysis
2. Read/write conflicts
3. Concurrency control techniques
4. Locking
   - Locking protocol
   - How locking influence concurrency problem
   - Deadlock
5. Deadlock

Classification of database systems

- One criteria is how many users can access the system concurently:
  - *Single-user* - if only one user can access the database system at one point of time.
  - *Multi-user* - if more users can access the database concurrently.
- We speak about the concurrency if more people access the RDBMS at one point of time.
- Concurrency brings a lot of problems, which influence also the programmers who access the database.

Transaction processing plans

- *Transaction plan* is a sequence of database operations performed during a transaction. We will use these plans to demonstrate problems of concurrency control.
- If we process several plans in one moment, we speak about *concurrent plan* or *parallel plan*.
- To simplify a plan we use the following operations:
  - READ instead of SELECT,
  - WRITE instead of UPDATE, however, these operations can involve also INSERT and DELETE.
- Operations work with tuples (records).

Concurrency control problems

- *Lost update*,
- *Uncommitted dependency*,
- *Inconsistent analysis*.

Lost update

Let us consider these two parallel plans:

Transaction A	Time	Transaction B
`READ` t	t_1	-
-	t_2	`READ` t
`WRITE` t	t_3	-
-	t_4	`WRITE` t

1. Transaction A gets the tuple t in t_1 time.
2. Transaction B gets the tuple t in t_2 time.
3. Transaction A changes the content of the tuple in t_3.
4. Transaction B rewrites the content of the tuple by its own value in t_4.

$\Rightarrow$ We lost the update made by the transaction A in t_3 time.

Problem of uncommitted dependency 1/3

- The problem of uncommitted dependency occurs in a case when one transaction reads or updates a tuple which was updated by a transaction which is still not committed.
- Since the transaction is not committed, ROLLBACK can happen.
- In the case of ROLLBACK the first transaction works with values which are not valid.

Problem of uncommitted dependency 2/3

Transaction A	Time	Transaction B
-	t_1	`WRITE` t
`READ` t	t_2	-
-	t_3	`ROLLBACK`

1. Transaction A reads an uncommitted value of a tuple t made by the transaction B in time t_2.
2. We ROLLBACK the transaction B in t_3 time.
3. The transaction A then works with an invalid tuple t retrieved in time t_2. The value of t is the value before t_1.

Problem of uncommitted dependency 3/3

Transaction A	Time	Transaction B
-	t_1	`WRITE` t
`WRITE` t	t_2	-
-	t_3	`ROLLBACK`

1. The transaction A writes the tuple t in t_2 time and becomes dependent on an uncommitted change from t_1 time.
2. We ROLLBACK the transaction B in t_3 time.
3. We lost the change made by the transaction A in t_2 time, since we ROLLBACK to the value of t in time before t_1.

Problem of inconsistent analysis 1/3

Transactions A and B work with accounts acc_1, acc_2 a acc_3. Accounts store values 30, 20, 50 before t_1.
The transaction A sums the account values, the transaction B moves the value 10 from account acc_1 to account acc_3.

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transaction A	**Time**	**Transaction B**
`READ` acc_1	t_1	-
$suma = 30$		
`READ` acc_2	t_2	-
$suma = 50$		
-	t_3	`READ` acc_3
-	t_4	`WRITE` $acc_3 = 60$
-	t_5	`READ` acc_1
-	t_6	`WRITE` $acc_1 = 20$
-	t_7	`COMMIT`
`READ` acc_3	t_8	-
$suma = 110$ but it should be 100		

Problem of inconsistent analysis 2/3

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transaction A	**Time**	**Transaction B**
`READ` acc_1	t_1	-
$suma = 30$		
`READ` acc_2	t_2	-
$suma = 50$		
-	t_3	`READ` acc_3
-	t_4	`WRITE` $acc_3 = 60$
-	t_5	`READ` acc_1
-	t_6	`WRITE` $acc_1 = 20$
-	t_7	`COMMIT`
`READ` acc_3	t_8	-
$suma = 110$ but it should be 100		

Transaction A works with inconsistent database and therefore it makes an inconsistent analysis (sum 110 instead of 100).

Problem of inconsistent analysis 3/3

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transaction A	**Time**	**Transaction B**
`READ` acc_1	t_1	-
$suma = 30$		
`READ` acc_2	t_2	-
$suma = 50$		
-	t_3	`READ` acc_3
-	t_4	`WRITE` $acc_3 = 60$
-	t_5	`READ` acc_1
-	t_6	`WRITE` $acc_1 = 20$
-	t_7	`COMMIT`
`READ` acc_3	t_8	-
$suma = 110$ but it should be 100		

If we compare it with the problem of uncommitted dependency, we see that the transaction A is not dependent on uncommitted changes of the transaction B since the B committed all changes before A read the acc_3.

Read/write conflicts

- Concurrency problem occurs when two transactions want to read or write the same tuple.
- There are four different types of conflicts: **RR** (READ-READ), **RW**, **WR**, **WW**.
- In a case of the RR conflict, transactions A and B want to read t. Two read operations can not influence each other, therefore, there is not such a problem.

RW conflict 1/2

- A reads t and B wants to write t. If B proceeds the write, then *problem of inconsistent analysis* can occur.
- The problem of inconsistent analysis is caused by the RW conflict.

Transaction A	**Time**	**Transaction B**
`READ` t	t_1	
	t_2	`WRITE` t
(another computations)	t_3	

RW conflict 2/2

- If B writes t and A reads t again, then A receives different values.
- We call such a situation *nonrepeatable read* – it is caused by the RW conflict.

Transaction A	Time	Transaction B
`READ` t	t_1	
	t_2	`WRITE` t
`READ` t	t_3	

WR conflict

- A writes t and B wants to read t.
- If B reads t, the **problem of uncommitted dependency** can occur.
- It is called *dirty read*.

Transaction A	Time	Transaction B
`WRITE` t	t_1	
	t_2	`READ` t
ROLLBACK ?	t_3	

WW conflict

- A writes t and B wants to write t.
- If B writes t, **the lost update problem** or **problem of uncommitted dependency** can occur (in the case of the transaction B).
- It is called *dirty write*.

Transaction A	Time	Transaction B
WRITE t	t_1	
	t_2	WRITE t
ROLLBACK ?	t_3	

Concurrency control techniques

- There is a number of concurrency control techniques which try to avoid the problems mentioned[1]:
    - **locking**,
    - **multi-versioning**,
    - *timestamps*,
    - *validation*.

[1] Later we will speak about the serializability of transactions.

Concurrency control techniques

- **Locking** – a pessimistic concurrency control technique: conflicts of parallel transactions are expected.
  - A DBMS manages one copy of data and locks are assigned to transactions.
- **Multi-versioning** – an optimistic concurrency control technique: conflicts of parallel transactions are not expected.
  - A copy of data is created for each transaction, a DBMS should manage which copy is valid.

Locking

- **Locking principle**: If a transaction A wants to read or write a tuple in a database, it asks for a lock.
- Another transaction which want to read or write the same tuple has to wait until the transaction A is going to release the lock.

Types of locks

There are two types of locks:

1. *Exclusive lock* (or *write lock*), we denote X.
2. *Shared lock* (or *read lock*), we denote S.

There are also other types of locks, however, we consider only these two types to keep it simple now.

Types of locks

The locks are assigned using these rules:

1. If a transaction A keeps the **exclusive lock** on a tuple t, then a request of a parallel transaction B on any type of a lock on the same tuple is not processed until A is not going to release the lock. It means, B **is waiting**.

2. If a transaction A keeps the **shared lock** on a tuple t, then a request of a parallel transaction B on:
    1. **An exclusive lock** is not processed until A is not going to release the lock. It means, B **is waiting**.
    2. **A shared lock** is processed immediately. It means, both transactions **keep the shared lock** on t.

Compatibility matrix for a X and S locks

- We can describe these rules using *lock type compatibility matrix*.
- The matrix can be interpreted as follows:
  - Let us consider the tuple t and transaction A which holds the lock corresponding to the first row of the table.
  - A parallel transaction B requests the lock on the tuple t, where the lock type is in the first column of the table.
- Symbol N stands for conflict (tuple t is not locked by the transaction B immediately).

	X	S	-
X	N	N	A
S	N	A	A
-	A	A	A

Remarks

- Locks are usually requested **implicitly**:
  - The **S** lock is requested automatically before a transaction gets a tuple from a database (for example using the `SELECT` command).
  - The **X** lock is requested before the transaction changes the tuple.
- The change of a tuple can be processed by: `UPDATE`, `INSERT` or `DELETE` operations[2].

[2] There are some differences, but we can ignore them now.

Locking protocol 1/3

To solve conflicts of concurrency it is necessary to define the *data access protocol* (or *locking protocol*):

1. A transaction which wants to **read a tuple**, has to first request **the shared lock** on this tuple.
2. A transaction which wants to **write a tuple**, has to first request **the exclusive lock** on this tuple. If the transaction holds the S lock then this lock is changed to the X lock.

Locking protocol 2/3

3. If the lock requested by the transaction B **can not be granted immediately**:
   - The transaction B is switched into **a wait state**.
   - The transaction remains in this state until the request is not processed, i.e. it waits for the release of locks handled by other transactions.
   - A DBMS should prevent the wait state of a transaction forever – this issue is called **livelock** or **starvation**.
   - The most simple way how to solve this issue is to keep **a queue of the requests**: the first transaction asking for the lock first gets it and so on.

Locking protocol 3/3

3. **The exclusive locks** are automatically released at the end of the transaction (finished with COMMIT or ROLLBACK operations). **The shared locks** are usually also released at the end of the transaction.

This protocol is called the **strict two-phase locking**.

Locking – Lost update problem 1/3

Transaction A	Time	Transaction B
READ t (S lock on t assigned)	t_1	-
-	t_2	READ t (S lock on t assigned)
WRITE t (X lock on t requested)	t_3	-
wait	t_4	WRITE t (X lock on t requested)
wait		wait
wait		wait

- The write of a tuple t by the transaction A in time t_3 is not accepted due to its request for the X lock which goes against the S lock of the transaction B.

Locking – Lost update problem 2/3

Transaction A	Time	Transaction B
READ t	t_1	-
(S lock on t assigned)		
-	t_2	READ t
		(S lock on t assigned)
WRITE t	t_3	-
(X lock on t requested)		
wait	t_4	WRITE t
		(X lock on t requested)
wait		wait
wait		wait

- The transaction A is switched into **the wait state** in t_3. Due to the same reason also the transaction B is switched into the wait state in t_4. Therefore, both transactions idle.

Locking – Lost update problem 3/3

Transaction A	Time	Transaction B
READ t (S lock on t received)	t_1	-
-	t_2	READ t (S lock on t received)
WRITE t (X lock on t requested)	t_3	-
wait	t_4	WRITE t (X lock on t requested)
wait		wait
wait		wait

- We see that we solve one problem, however, another problem occurs. We call such a problem as **deadlock**.

Locking – Uncommitted dependency I 1/3

Transaction A	Time	Transaction B
-	t_1	`WRITE` t (X lock on t assigned)
`READ` t (S lock on t requested)	t_2	-
wait	t_3	`COMMIT/ROLLBACK` (X lock on t released
repeat: `READ` t (S lock on t assigned)	t_4	

- The read of the transaction A in t_2 is not accepted due to the conflict with the X lock assigned to the transaction B in t_1.
- The transaction A is switched into the wait state until the transaction B is finished by COMMIT or ROLLBACK and releases the exclusive lock on t.

Locking – Uncommitted dependency I 2/3

Transaction A	Time	Transaction B
-	t_1	`WRITE` t (X lock on t assigned)
`READ` t (S lock on t requested)	t_2	-
wait	t_3	`COMMIT/ROLLBACK` (X lock on t released
repeat: `READ` t (S lock on t assigned)	t_4	

- After the release, A continues: it works with the valid value of t: if the transaction B proceeds COMMIT, then the transaction A works with the value set in t_1, if B proceeds ROLLBACK then A works with the value set before t_1.

Locking – Uncommitted dependency I 3/3

Transaction A	Time	Transaction B
-	t_1	`WRITE` t (X lock on t assigned)
`READ` t (S lock on t requested)	t_2	-
wait	t_3	`COMMIT/ROLLBACK` (X lock on t released
repeat: `READ` t (S lock on t assigned)	t_4	

- Since the transaction A is not dependent on an uncommitted update of the transaction B, this problem is solved.

Locking – Uncommitted dependency II 1/3

Transaction A	Time	Transaction B
-	t_1	WRITE t (X lock on t assigned)
WRITE t (X lock on t requested)	t_2	-
wait	t_3	COMMIT/ROLLBACK (X lock on t released)
repeat: WRITE t (X lock on t assigned)	t_4	

- The write operation of the transaction A in t_2 is not accepted due to the conflict with the X lock of the transaction B.
- Therefore, the transaction A is switched into the wait state, until the transaction B is finished by COMMIT or ROLLBACK and releases the X lock on t.

Locking – Uncommitted dependency II 2/3

Transaction A	Time	Transaction B
-	t_1	`WRITE` t (X lock on t assigned)
`WRITE` t (X lock on t requested)	t_2	-
wait	t_3	`COMMIT/ROLLBACK` (X lock on t released)
repeat: `WRITE` t (X lock on t assigned)	t_4	

- The transaction A continues, it receives the X lock on t with the valid value: if B proceeds COMMIT, then A has the value set in t_1, if B proceeds ROLLBACK, then A works with the value set before t_1.

Locking – Uncommitted dependency II 3/3

Transaction A	Time	Transaction B
-	t_1	WRITE t (X lock on t assigned)
WRITE t (X lock on t requested)	t_2	-
wait	t_3	COMMIT/ROLLBACK (X lock on t released)
repeat: WRITE t (X lock on t assigned)	t_4	

- Since the transaction A is not dependent on an uncommitted update of the transaction B, this problem is solved.

Locking – Inconsistent analysis 1/3

$acc_1 = 30$	$acc_2 = 20$	$acc_3 = 50$
Transaction A	**Time**	**Transaction B**
READ acc_1 (S lock on acc_1 assigned) $suma = 30$	t_1	-
READ acc_2 (S lock on acc_2 assigned) $suma = 50$	t_2	-
-	t_3	READ acc_3 (S lock on acc_3 assigned)
-	t_4	WRITE $acc_3 = 60$ (X lock on acc_3 assigned)
-	t_5	READ acc_1 (S lock on acc_1 assigned)
...	...	...

Locking – Inconsistent analysis 2/3

$acc_1 = 30$	$acc_2 = 20$	$acc_3 = 50$
Transaction A	**Time**	**Transaction B**
...	...	...
-	t_6	`WRITE` $acc_1 = 20$
		(X lock on acc_1 requested)
`READ` acc_3	t_7	wait
(S lock on acc_3 requested)		
wait		wait

- The write of acc_1 by the transaction B in t_6 is not accepted due to the conflict of the X lock request with the S lock kept by the A transaction.
- The transaction B is switched into the wait state.

Locking – Inconsistent analysis 3/3

$acc_1 = 30$	$acc_2 = 20$	$acc_3 = 50$
Transaction A	**Time**	**Transaction B**
...	...	...
-	t_6	WRITE $acc_1 = 20$
		(X lock on acc_1 requested)
READ acc_3	t_7	wait
(S lock on acc_1 requested)		
wait		wait

- Similarly, the read of the transaction A in t_7 is not accepted since the conflict of the S lock request with the share lock of B. A is switched into the wait state.
- The original problem of inconsistent analysis is solved, however, **the deadlock occurs**.

Deadlock

It means the two-phase locking protocol can lead to a deadlock. In the following figure we see a general scheme of locking where the deadlock occurs.

Transaction A	Time	Transaction B
S lock on r_1 assigned	t_1	
-	t_2	S lock on r_2 assigned
X lock on r_2 requested	t_3	-
wait	t_4	X lock on r_1 requested
wait		wait

Remark: r_1 and r_2 are any database objects. They do not have to be tuples.

Deadlock

There are several techniques how to solve a deadlock:

1. **Deadlock detection**:
   1. Time limits for a wait time,
   2. Cycle detection in graph *Wait-For*.

2. **Avoiding deadlock**

Deadlock detection - time limits

- Is is assumed that a transaction takes **a limited time**.
- When the transaction takes longer time, a system supposes that the deadlock occurs.

Deadlock detection - cycle detection

- **A cycle detection** is a more effective type of the deadlock detection. It creates **a graph of transactions** waiting for each other.
- The deadlock is solved by a selection of one deadlocked transaction for which the **ROLLBACK is processed** (i.e. all locks of the transaction are released).
- Remaining transactions in the deadlock can continue.
- A DBMS processes the cancelled transaction again or it returns a deadlock exception.

Avoiding deadlock 1/3

This strategy tries to avoid the deadlock by a modification of a locking protocol. There are two versions of locking protocol called **wait-die** and **wound-wait**:

1. Each transaction gets a timestamp – the time of the transaction start – which is unique.

Avoiding deadlock 2/3

2. If the transaction A requests a lock on a tuple, which is already locked by the transaction B, then:
    1. **Wait-die variant**: if A is older than B, then A is switched into the wait state; if A is younger than B, then A is canceled using ROLLBACK and runs again[3].
    2. **Wound-wait variant**: if A is older than B, then the transaction B is cancelled using ROLLBACK and runs again[4], if A is younger than B, then A is switched into the wait state;
3. If the transaction runs again the time-stamp remains the same.

[3] Transaction A die.

[4] Transaction B is wounded.

Avoiding deadlock 3/3

- The first part of the variant name describes a situation when A is older than B.
  - In the case of **wait-die**, the older transactions are switched into the wait state,
  - In the case of **wound-wait**, the younger transactions are switched into the wait state.
- It has been proved that the deadlock can not occur in the case of these protocols.
- A disadvantage of these protocols is relatively a high number of ROLLBACK operations.