Data Base Systems

VU 184.686, WS 2020

Concurrency Control

Anela Lolić

Institute of Logic and Computation, TU Wien

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Acknowledgements

The slides are based on the slides (in German) of Sebastian Skritek.

The content is based on Chapter 11 of
(Kemper, Eickler: Datenbanksysteme – Eine Einführung).

For related literature in English see Chapter 17 of
(Ramakrishnan, Gehrke: Database Management Systems).

Concurrency Control

Overview

Overview

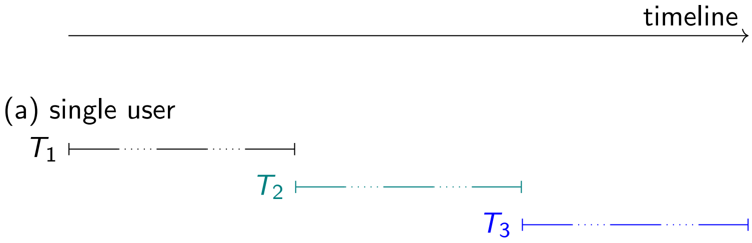
Concurrency

execution of three transaction T_1, T_2 and T_3:

$$\xrightarrow{\text{timeline}}$$

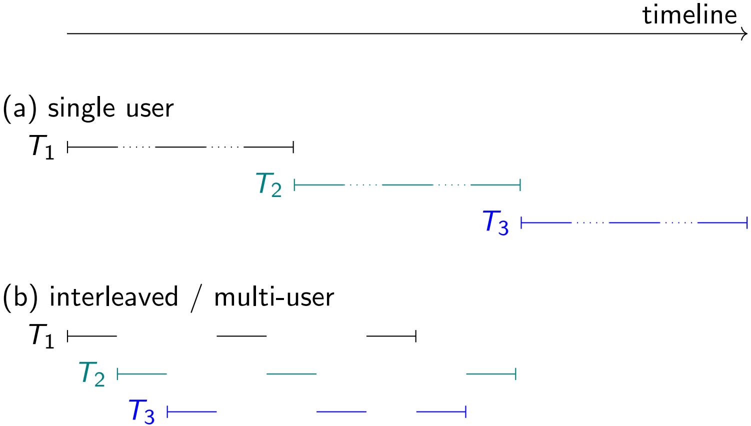
Concurrency

execution of three transaction T_1, T_2 and T_3:



Concurrency

execution of three transaction T_1, T_2 and T_3:



Parallel Execution

idea:

- CPU- and I/O-activities may be executed in parallel
- interleaved execution of several transactions leads to better resources utilization

Parallel Execution

idea:

- CPU- and I/O-activities may be executed in parallel
- interleaved execution of several transactions leads to better resources utilization

advantages:

- DBMS performance/throughput can be increased (average number of completed transactions per time unit)
- this might reduce unpredictable delays in response times

Overview

Possible Errors with uncontrolled Concurrency

- lost update
- dirty read
- unrepeatable read
- phantom problem

Lost Update

Example

T_1: transaction from A to B

T_2: interest credit for A

Lost Update

Example

T_1: transaction from A to B

T_2: interest credit for A

step	T_1	T_2
1.	read(A,a_1)	
2.		read(A,a_2)
3.		write(A,$a_2 * 1.03$)
4.	write(A,$a_1 - 300$)	
5.	read(B,b_1)	
6.	write(B,$b_1 + 300$)	

Lost Update

Example

T_1: transaction from A to B

T_2: interest credit for A

step	T_1	T_2
1.	read(A,a_1)	
2.		read(A,a_2)
3.		write(A,$a_2 * 1.03$)
4.	write(A,$a_1 - 300$)	
5.	read(B,b_1)	
6.	write(B,$b_1 + 300$)	

problem: T_1 overwrites the modification performed by T_2

$\Rightarrow$ the modifications of T_2 get lost

Dirty Read

reading of modifications that were not commited

Example

step	T_1	T_2
1.	read(A,a_1)	
2.	write(A,$a_1 - 300$)	
3.		read(A,a_2)
4.		write(A,$a_2 * 1.03$)
5.	read(B,b_1)	
6.	...	
7.	**abort**	

Dirty Read

reading of modifications that were not commited

Example

step	T_1	T_2
1.	read(A,a_1)	
2.	write(A,$a_1 - 300$)	
3.		read(A,a_2)
4.		write(A,$a_2 * 1.03$)
5.	read(B,b_1)	
6.	...	
7.	**abort**	

problem:

- changes in T_2 are based on an inconsistent database instance
- even for `commit` instead of `abort`

Unrepeatable Read

Example

step	T_1	T_2
1.	read(A,a_1)	
2.	...	
3.		read(A,a_2)
4.		write(A,$a_2 * 1.03$)
5.		**commit**
6.	read(A,a_1)	
7.	...	

Unrepeatable Read

Example

step	T_1	T_2
1.	read(A, a_1)	
2.	...	
3.		read(A, a_2)
4.		write$(A, a_2 * 1.03)$
5.		**commit**
6.	read(A, a_1)	
7.	...	

problem: repeated reading by T_1 leads to different results, although T_1 has not performed any modifications

Phantom Problem

Example

step	T_1	T_2
1.		`SELECT SUM(accountBalance)` `FROM Konten`
2.	`INSERT INTO Konten` `VALUES (C,1000,...)`	
3.		`SELECT SUM(accountBalance)` `FROM Konten`

Phantom Problem

Example

step	T_1	T_2
1.		`SELECT SUM(accountBalance)` `FROM Konten`
2.	`INSERT INTO Konten` `VALUES (C,1000,...)`	
3.		`SELECT SUM(accountBalance)` `FROM Konten`

problem: T_2 computes different values, as “phantom” with values $(C, 1000, \dots)$ was inserted

Phantom Problem

Example

step	T_1	T_2
1.		`SELECT SUM(accountBalance)` `FROM Konten`
2.	`INSERT INTO Konten` `VALUES (C,1000,...)`	
3.		`SELECT SUM(accountBalance)` `FROM Konten`

problem: T_2 computes different values, as "phantom" with values $(C, 1000, \ldots)$ was inserted

at access to sets of objects with specific feature

Conflicts Between Transactions

conflict:

- two transactions want to access the same object
- at least one of the accesses is a writing access
- possible conflicts: W-W, W-R, R-W

Conflicts Between Transactions

conflict:

- two transactions want to access the same object
- at least one of the accesses is a writing access
- possible conflicts: W-W, W-R, R-W

errors caused by conflicts:

- lost update: W-W
- dirty read: W-R
- unrepeatable read: R-W
- phantom problem: R-W

Overview

Overview

Transactions

transaction: consists of a set of operations as well as a (partial) order on these operations

Transactions

transaction: consists of a set of operations as well as a (partial) order on these operations

elementary operations of a transaction T_i:

$r_i(A)$: reads data object A

$w_i(A)$: writes data object A

a_i: abort of transaction

c_i: commit of transaction

(we do not consider insert or delete)

Transactions

transactions: consists of a set of operations as well as a (partial) order on these operations

Transactions

transactions: consists of a set of operations as well as a (partial) order on these operations

minimum requirement for the **order** $<_i$ on operations:

- in case the transaction contains `abort`:
  $o_i(A) <_i a_i$ for all operations $o_i(A)$ (`abort` last action)
- in case the transaction contains `commit`:
  $o_i(A) <_i c_i$ for all operations $o_i(A)$ (`commit` last action)
- has to be defined on all pairs of operations on the same data object

in general: total order

Schedules

schedule: temporal order of elementary operations of a set of transactions

Schedules

schedule: temporal order of elementary operations of a set of transactions

features of a schedule S with order $<_S$:

- S contains exactly the elementary operations of all involved transactions
- $<_S$ is compatible with all $<_i$
- for conflict operations p, q it holds that either $p <_S q$ or $q <_S p$

Schedules

schedule: temporal order of elementary operations of a set of transactions

features of a schedule S with order $<_S$:

- S contains exactly the elementary operations of all involved transactions
- $<_S$ is compatible with all $<_i$
- for conflict operations p, q it holds that either $p <_S q$ or $q <_S p$

conflict operations: pairs of operations that access the same data object and at least one of them is a writing operation:

Schedules

schedule: temporal order of elementary operations of a set of transactions

features of a schedule S with order $<_S$:

- S contains exactly the elementary operations of all involved transactions
- $<_S$ is compatible with all $<_i$
- for conflict operations p, q it holds that either $p <_S q$ or $q <_S p$

conflict operations: pairs of operations that access the same data object and at least one of them is a writing operation:
$(r_i(A), w_j(A)), \quad (w_i(A), r_j(A)), \quad (w_i(A), w_j(A))$

Schedule for Three Transactions

Example

$$S = \begin{tikzcd}
 & r_2(A) \arrow[r] & w_2(B) \arrow[r] & w_2(C) \arrow[r] & c_2 \\
r_3(B) \arrow[r] & w_3(A) \arrow[r] \arrow[u] & w_3(B) \arrow[r] \arrow[u] & w_3(C) \arrow[r] \arrow[u] & c_3 \\
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[u] & c_1 & &
\end{tikzcd}$$

Overview

Serializability

serial schedule: each transaction is completely executed before the next one starts

notation: $T_1 \mid T_2 \mid T_3 \ldots$ (for "T_1 before T_2 before $T_3 \ldots$").

Serializability

serial schedule: each transaction is completely executed before the next one starts

notation: $T_1 \mid T_2 \mid T_3 \ldots$ (for "T_1 before T_2 before T_3 ...").

serializable schedule: a parallel schedule with same effect as a serial execution (informally)

Serializability

serial schedule: each transaction is completely executed before the next one starts

notation: $T_1 \mid T_2 \mid T_3 \ldots$ (for "T_1 before T_2 before T_3 ...").

serializable schedule: a parallel schedule with same effect as a serial execution (informally)

- on each database instance
- from the perspective of the DBMS; DBMS only sees the elementary operations (`read`, `write`, ...) but does not see the application logic

Serializable Schedule

Example

step	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.		**BOT**
4.		read(C)
5.	write(A)	
6.		write(C)
7.	read(B)	
8.	write(B)	
9.	**commit**	
10.		read(A)
11.		write(A)
12.		**commit**

Equivalent Serial Execution: $T_1 \mid T_2$

Example

step	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.	read(B)	
5.	write(B)	
6.	**commit**	
7.		**BOT**
8.		read(C)
9.		write(C)
10.		read(A)
11.		write(A)
12.		**commit**

Not Serializable Schedule

Example

step	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.		**BOT**
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		**commit**
10.	read(B)	
11.	write(B)	
12.	**commit**	

Remark: Serializability Always from the Perspective of a DBMS

- the last schedule is not serializable from the perspective of the DBMS
- DBMS does not see any application logic
- based on the application logic it can be that . . .

Remark: Serializability Always from the Perspective of a DBMS

- the last schedule is not serializable from the perspective of the DBMS
- DBMS does not see any application logic
- based on the application logic it can be that ...
  - ... two transactions fitting the schedule would have the same effect at a serial execution (see the next example)

FAKULTÄT FÜR INFORMATIK
Faculty of Informatics

Remark: Serializability Always from the Perspective of a DBMS

- the last schedule is not serializable from the perspective of the DBMS
- DBMS does not see any application logic
- based on the application logic it can be that ...
  - ... two transactions fitting the schedule would have the same effect at a serial execution (see the next example)
  - ... for two transactions fitting the schedule there is no serial execution with the same effect (see next but one example)

Serializable: Two Bank Transactions

Example

step	T_1	T_2
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
6.		read(A,a_2)
7.		$a_2 := a_2 - 100$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 + 100$
11.		write(B,b_2)
12.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

Serializable: Two Bank Transactions

Example

step	T_1	T_2
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
6.		read(A,a_2)
7.		$a_2 := a_2 - 100$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 + 100$
11.		write(B,b_2)
12.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

A Transaction and an Interest Credit

Example

step	T_1	T_2
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
6.		read(A,a_2)
7.		$a_2 := a_2 * 1.03$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 * 1.03$
11.		write(B,b_2)
12.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

A Transaction and an Interest Credit

Example

step	T_1	T_2
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
6.		read(A,a_2)
7.		$a_2 := a_2 * 1.03$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 * 1.03$
11.		write(B,b_2)
12.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

Serializable Schedules

Definition (serializable (no aborted transactions))

A schedule over a set of successfully completed transactions is *serializable* if its effect to any consistent data base instance is identical to the effect of a serial schedule of the same transaction.

Serializable Schedules

Definition (serializable (no aborted transactions))

A schedule over a set of successfully completed transactions is *serializable* if its effect to any consistent data base instance is identical to the effect of a serial schedule of the same transaction.

Definition (serializable (with aborted transactions))

A schedule over a set of transactions is *serializable* if its effect to any consistent data base instance is identical to a serial schedule of successful (`committed`) transactions.

Conflict Serializable Schedules

find: possibility to decide:

- Is a schedule serializable?
- Which serial execution has the same result?

Conflict Serializable Schedules

find: possibility to decide:

- Is a schedule serializable?
- Which serial execution has the same result?

$\Rightarrow$ conflict serializable schedules

Conflict Serializable Schedules

Definition (conflict equivalent)

Two schedules (over the same set of transactions) are *conflict equivalent* if they execute all conflict operations of not aborted transactions in the same order.

Conflict Serializable Schedules

Definition (conflict equivalent)

Two schedules (over the same set of transactions) are *conflict equivalent* if they execute all conflict operations of not aborted transactions in the same order.

Example (conflict equivalent schedules)

$r_1(A) \rightarrow r_2(C) \rightarrow w_1(A) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

Conflict Serializable Schedules

Definition (conflict equivalent)

Two schedules (over the same set of transactions) are *conflict equivalent* if they execute all conflict operations of not aborted transactions in the same order.

Example (conflict equivalent schedules)

$$r_1(A) \to r_2(C) \to w_1(A) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_2(C) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to r_2(C) \to w_2(C) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to w_1(B) \to c_1 \to r_2(C) \to w_2(C) \to r_2(A) \to w_2(A) \to c_2$$

Conflict Serializable Schedules

Definition (conflict equivalent)

Two schedules (over the same set of transactions) are *conflict equivalent* if they execute all conflict operations of not aborted transactions in the same order.

Example (conflict equivalent schedules)

$r_1(A) \rightarrow r_2(C) \rightarrow w_1(A) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

Conflict Serializable Schedules

Definition (conflict equivalent)

Two schedules (over the same set of transactions) are *conflict equivalent* if they execute all conflict operations of not aborted transactions in the same order.

Example (conflict equivalent schedules)

$$r_1(A) \to r_2(C) \to w_1(A) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_2(C) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to r_2(C) \to w_2(C) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to w_1(B) \to c_1 \to r_2(C) \to w_2(C) \to r_2(A) \to w_2(A) \to c_2$$

Conflict Serializable Schedules

Definition (conflict equivalent)

Two schedules (over the same set of transactions) are *conflict equivalent* if they execute all conflict operations of not aborted transactions in the same order.

Example (conflict equivalent schedules)

$r_1(A) \rightarrow r_2(C) \rightarrow w_1(A) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

Conflict Serializable Schedules

Definition (conflict serializable)

A schedule is *conflict serializable* if it is conflict equivalent to a serial schedule.

Conflict Serializable Schedules

Definition (conflict serializable)

A schedule is *conflict serializable* if it is conflict equivalent to a serial schedule.

Example (conflict serializable schedule)

$$
S = \begin{tikzcd}
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[ddl] & w_1(B) \arrow[r] & c_1 \\
 & r_2(A) \arrow[u] \arrow[r] \arrow[d] & w_2(B) \arrow[u] \arrow[r] & c_2 \\
r_3(A) \arrow[r] & w_3(A) \arrow[r] & c_3 &
\end{tikzcd}
$$

Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \to T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \rightarrow T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$$
\begin{tikzcd}
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[ddl] & w_1(B) \arrow[r] & c_1 \\
& r_2(A) \arrow[u] \arrow[r] \arrow[d] & w_2(B) \arrow[u] \arrow[r] & c_2 \\
r_3(A) \arrow[r] & w_3(A) \arrow[r] & c_3 &
\end{tikzcd}
$$

Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \rightarrow T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$$
\begin{tikzcd}
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[dl] & w_1(B) \arrow[r] & c_1 \\
 & r_2(A) \arrow[r] \arrow[u] \arrow[d] & w_2(B) \arrow[r] \arrow[u] & c_2 \\
r_3(A) \arrow[r] & w_3(A) \arrow[r] & c_3 &
\end{tikzcd}
$$

T_1

T_2 T_3

Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \rightarrow T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$$
\begin{tikzcd}
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[ddl] & w_1(B) \arrow[r] & c_1 \\
 & r_2(A) \arrow[r] \arrow[u] \arrow[d] & w_2(B) \arrow[r] \arrow[u] & c_2 \\
r_3(A) \arrow[r] & w_3(A) \arrow[r] & c_3 &
\end{tikzcd}
$$

T_1

T_2 T_3

Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \rightarrow T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$r_1(A) \longrightarrow w_1(A) \longrightarrow w_1(B) \longrightarrow c_1$

$r_2(A) \longrightarrow w_2(B) \longrightarrow c_2$

$r_3(A) \longrightarrow w_3(A) \longrightarrow c_3$

$T_2 \rightarrow T_1$

T_3

Serializability Graph

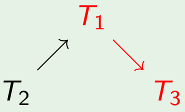
Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \rightarrow T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$$
\begin{array}{ccccccc}
r_1(A) & \longrightarrow & w_1(A) & \longrightarrow & w_1(B) & \longrightarrow & c_1 \\
 & & \uparrow & & \uparrow & & \\
 & & r_2(A) & \longrightarrow & w_2(B) & \longrightarrow & c_2 \\
 & & \downarrow & & & & \\
r_3(A) & \longrightarrow & w_3(A) & \longrightarrow & c_3 & &
\end{array}
$$



Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \to T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$$
\begin{tikzcd}
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[ddl] & w_1(B) \arrow[r] & c_1 \\
 & r_2(A) \arrow[r] \arrow[u] \arrow[d] & w_2(B) \arrow[r] \arrow[u] & c_2 \\
r_3(A) \arrow[r] & w_3(A) \arrow[r] & c_3 &
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & \color{red}{T_1} \arrow[dr] & \\
T_2 \arrow[ur] & & \color{red}{T_3}
\end{tikzcd}
$$

Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \rightarrow T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$$\begin{tikzcd}
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[ddl] & w_1(B) \arrow[r] & c_1 \\
& r_2(A) \arrow[u] \arrow[r] \arrow[d] & w_2(B) \arrow[u] \arrow[r] & c_2 \\
r_3(A) \arrow[r] & w_3(A) \arrow[r] & c_3 &
\end{tikzcd}$$

$$\begin{tikzcd}
& T_1 \arrow[dr] & \\
T_2 \arrow[ur] \arrow[rr] & & T_3
\end{tikzcd}$$

Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \to T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$$
\begin{tikzcd}
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[ddl] & w_1(B) \arrow[r] & c_1 \\
 & r_2(A) \arrow[u] \arrow[r] \arrow[d] & w_2(B) \arrow[u] \arrow[r] & c_2 \\
r_3(A) \arrow[r] & w_3(A) \arrow[r] & c_3 &
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & T_1 \arrow[dr] & \\
T_2 \arrow[ur] \arrow[rr] & & T_3
\end{tikzcd}
$$

Serializability Graph

Definition (serializability graph)

serializability graph $SG(S)$ of a schedule S:

- nodes: successful transactions $T_1, T_2, \ldots$ of S
- directed edge $T_i \rightarrow T_j$ if for (at least) one pair (p_i, q_j) of conflict operations it holds that: $p_i <_S q_j$

Example

$$
\begin{tikzcd}
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[ddl] & w_1(B) \arrow[r] & c_1 \\
 & r_2(A) \arrow[u] \arrow[d] \arrow[r] & w_2(B) \arrow[u] \arrow[r] & c_2 \\
r_3(A) \arrow[r] & w_3(A) \arrow[r] & c_3 &
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & T_1 \arrow[dr] & \\
T_2 \arrow[ur] \arrow[rr] & & T_3
\end{tikzcd}
$$

Serializability Theorem

Theorem

A schedule S is conflict serializable if and only if the corresponding serializability graph $SG(S)$ is acyclic.

Serializability Theorem

Theorem

A schedule S is conflict serializable if and only if the corresponding serializability graph $SG(S)$ is acyclic.

Theorem

Any topological sorting of $SG(S)$ specifies a conflict equivalent serial schedule.

Serializability Theorem

Theorem

A schedule S is conflict serializable if and only if the corresponding serializability graph $SG(S)$ is acyclic.

Theorem

Any topological sorting of $SG(S)$ specifies a conflict equivalent serial schedule.

Example

$S = w_1(A) \to w_1(B) \to c_1 \to r_2(A) \to r_3(B) \to w_2(A) \to c_2 \to w_3(B) \to c_3$

Serializability Theorem

Theorem

A schedule S is conflict serializable if and only if the corresponding serializability graph $SG(S)$ is acyclic.

Theorem

Any topological sorting of $SG(S)$ specifies a conflict equivalent serial schedule.

Example

$S = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

$$SG(S) = \begin{array}{ccc} & & T_2 \\ T_1 & & \\ & & T_3 \end{array}$$

Serializability Theorem

Theorem

A schedule S is conflict serializable if and only if the corresponding serializability graph $SG(S)$ is acyclic.

Theorem

Any topological sorting of $SG(S)$ specifies a conflict equivalent serial schedule.

Example

$S = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

$$SG(S) = \begin{tikzcd} & T_2 \\ T_1 \arrow[ur] & \\ & T_3 \end{tikzcd}$$

Serializability Theorem

Theorem

A schedule S is conflict serializable if and only if the corresponding serializability graph $SG(S)$ is acyclic.

Theorem

Any topological sorting of $SG(S)$ specifies a conflict equivalent serial schedule.

Example

$$S = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$$

$$SG(S) = \begin{tikzcd} & T_2 \\ T_1 \arrow[ur] \arrow[dr] & \\ & T_3 \end{tikzcd}$$

Serializability Theorem

Theorem

A schedule S is conflict serializable if and only if the corresponding serializability graph $SG(S)$ is acyclic.

Theorem

Any topological sorting of $SG(S)$ specifies a conflict equivalent serial schedule.

Example

$S = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

$$SG(S) = \begin{array}{ccc} & & T_2 \\ & \nearrow & \\ T_1 & & \\ & \searrow & \\ & & T_3 \end{array}$$

$$S_s^1 = T_1 \mid T_2 \mid T_3$$
$$S_s^2 = T_1 \mid T_3 \mid T_2$$
$$S \equiv S_s^1 \equiv S_s^2$$

Computation of a Topological Sorting

- observation: in an acyclic directed graph there is at least one node with no incoming edge

Computation of a Topological Sorting

- observation: in an acyclic directed graph there is at least one node with no incoming edge
- algorithm:

In: acyclic serilizability graph $SG(S)$
Out: a possible topological sorting

1: pick a node T_i without incoming edge in $SG(S)$
2: write T_i to the output
3: delete T_i and all outgoing edges from $SG(S)$
4: **if** there is still a node left **then**
5: **goto** 1
6: **end if**

Overview

More Features of Schedules

in case no transaction aborts:

- each conflict serializable schedule is serializable

More Features of Schedules

in case no transaction aborts:

- each conflict serializable schedule is serializable

attention: the opposite does not have to hold:

More Features of Schedules

in case no transaction aborts:

- each conflict serializable schedule is serializable

attention: the opposite does not have to hold:

Example

	T_1	T_2	T_3
1.	read(A)		
2.		write(A)	
3.		**commit**	
4.	write(A)		
5.	**commit**		
6.			write(A)
7.			**commit**

More Features of Schedules

if transactions may abort:

- not every conflict serializable schedule is serializable

More Features of Schedules

if transactions may abort:

- not every conflict serializable schedule is serializable

Example

	T_1	T_2
1.	read(A)	
2.	write(A)	
3.		read(A)
4.		write(A)
5.		read(B)
6.		write(B)
7.		**commit**
8.	**abort**	

Schedules at aborted Transactions

conflict serializable in DBMS is usually the minimal requirement

Schedules at aborted Transactions

conflict serializable in DBMS is usually the minimal requirement

further features of/requirements to schedules:

- resettable schedules
- schedules without cascading reset
- strict schedules

resettable schedules

minimal requirement w.r.t. recovery:

- abortion of an active transaction possible without influencing successfully completed transactions

resettable schedules

minimal requirement w.r.t. recovery:

- abortion of an active transaction possible without influencing successfully completed transactions

resettable schedules (informally): a transaction may only be terminated with a `commit` after all transactions from which it has read are completed as well

resettable schedules

minimal requirement w.r.t. recovery:

- abortion of an active transaction possible without influencing successfully completed transactions

resettable schedules (informally): a transaction may only be terminated with a `commit` after all transactions from which it has read are completed as well

Write/Read Dependencies between Schedules

Definition (T_i reads from T_j)

A transaction T_i *reads from a transaction* T_j in a schedule S if there is at least one data object A such that the following holds:

Write/Read Dependencies between Schedules

Definition (T_i reads from T_j)

A transaction T_i *reads from a transaction* T_j in a schedule S if there is at least one data object A such that the following holds:

1. $w_j(A) <_S r_i(A)$:
   T_j writes at least a data object A that is read by T_i later on

Write/Read Dependencies between Schedules

Definition (T_i reads from T_j)

A transaction T_i *reads from a transaction* T_j in a schedule S if there is at least one data object A such that the following holds:

1. $w_j(A) <_S r_i(A)$:
   T_j writes at least a data object A that is read by T_i later on
2. $a_j \not<_S r_i(A)$:
   T_j is not reset before T_i has read data object A

Write/Read Dependencies between Schedules

Definition (T_i reads from T_j)

A transaction T_i *reads from a transaction* T_j in a schedule S if there is at least one data object A such that the following holds:

1. $w_j(A) <_S r_i(A)$:
   T_j writes at least a data object A that is read by T_i later on
2. $a_j \not<_S r_i(A)$:
   T_j is not reset before T_i has read data object A
3. if there exists a $w_k(A)$ with $w_j(A) <_S w_k(A) <_S r_i(A)$, then also $a_k <_S r_i(A)$:
   all intermediate writing processes of other transactions to A are reset before the reading process of T_i.

Write/Read Dependencies between Schedules

Definition (T_i reads from T_j)

A transaction T_i *reads from a transaction* T_j in a schedule S if there is at least one data object A such that the following holds:

1. $w_j(A) <_S r_i(A)$:
   T_j writes at least a data object A that is read by T_i later on
2. $a_j \not<_S r_i(A)$:
   T_j is not reset before T_i has read data object A
3. if there exists a $w_k(A)$ with $w_j(A) <_S w_k(A) <_S r_i(A)$, then also $a_k <_S r_i(A)$:
   all intermediate writing processes of other transactions to A are reset before the reading process of T_i.

intuition: T_i reads exactly the value for A written by T_j

Resettable Schedules

Definition (resettable schedules)

A schedule S is *resettable* if for all pairs T_i, T_j of transactions in S it holds that:

- if T_i reads from T_j and T_i executes a commit, then this must have been preceded by the commit of T_j (T_i reads from T_j and $c_i \Rightarrow c_j <_S c_i$).

Resettable Schedules

Definition (resettable schedules)

A schedule S is *resettable* if for all pairs T_i, T_j of transactions in S it holds that:

- if T_i reads from T_j and T_i executes a `commit`, then this must have been preceded by the `commit` of T_j (T_i reads from T_j and $c_i \Rightarrow c_j <_S c_i$).

alternatively: a transaction is only allowed to execute its `commit` when all transactions it has read from are terminated

Problem: Cascading Reset

Example (schedule with cascading reset)

step	T_1	T_2	T_3	T_4	T_5
0.	...				
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_4(D)$	
8.					$r_5(D)$
9.	a_1 (**abort**)				

Schedules Without Cascading Reset

Definition (schedule without cascading reset)

A schedule S *avoids cascading resets* if

- $c_j <_S r_i(A)$ hold whenever T_i reads a data object A from T_j (T_i reads from T_j only after T_j was `committed`).

Schedules Without Cascading Reset

Definition (schedule without cascading reset)

A schedule S *avoids cascading resets* if

- $c_j <_S r_i(A)$ hold whenever T_i reads a data object A from T_j (T_i reads from T_j only after T_j was `committed`).

alternatively: modifications by a transactions are released for reading only after a `commit`

Strict Schedules

Definition (strict schedule)

A schedule S is *strict* if for all pairs T_i, T_j of transactions in S it holds that:

- if $w_j(A) <_S o_i(A)$ (with $o_i \in \{r_i, w_i\}$), then either
  - $c_j <_S o_i(A)$ or
  - $a_j <_S o_i(A)$.

Strict Schedules

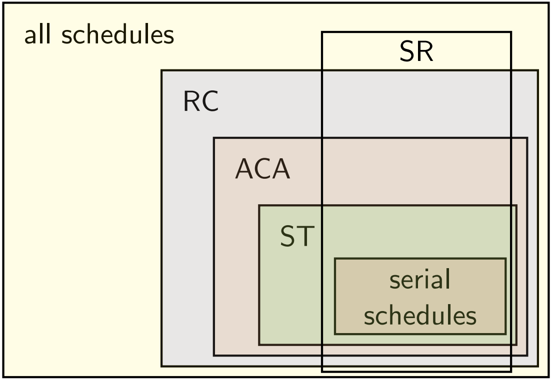
Definition (strict schedule)

A schedule S is *strict* if for all pairs T_i, T_j of transactions in S it holds that:

- if $w_j(A) <_S o_i(A)$ (with $o_i \in \{r_i, w_i\}$), then either
  - $c_j <_S o_i(A)$ or
  - $a_j <_S o_i(A)$.

alternatively: transactions get write or read access to a data object written by another transaction only after its completion (`commit` or `abort`)

Relationship Between the Classes of Schedules



- SR (conflict-) serializable schedules (*SeRializable*)
- RC resettable schedules (*ReCoverable*)
- ACA schedules without cascading reset (*avoid cascading abort*)
- ST strict schedules (*STrict*)

Overview

Database Scheduler

determines execution order of operations

- minimal requirement: resulting schedule should be conflict serializable
- usually only strict, conflict serializable schedules are constructed
- realization:
  - widespread: lock based synchronisation
  - other methods: timestamp based synch, optimistic sync, MVCC

Overview

Lock Based Synchronisation Protocols

- two-phase lock protocol (*2PL*)
- strict two-phase lock protocol (*strict 2PL*)
- two-phase lock protocol with preclaiming (*conservative 2PL*)
- lock protocols with timestamp
- hierarchical lock granulates (*MGL*)

Lock Modes

two lock modes:

S Shared, read lock

Lock Modes

two lock modes:

- S Shared, read lock
- X eXclusive, write lock

Lock Modes

two lock modes:

S Shared, read lock

X eXclusive, write lock

compatibility matrix

lock		current NL	current S	current X
requested	S	✓	✓	–
requested	X	✓	–	–

NL . . . *No Lock*

Two-Phase Lock Protocol (2PL)

each transaction considers the following rules:

Two-Phase Lock Protocol (2PL)

each transaction considers the following rules:

1. each object used by a transaction has to be locked before the use correspondingly

Two-Phase Lock Protocol (2PL)

each transaction considers the following rules:

1. each object used by a transaction has to be locked before the use correspondingly
2. a transaction does not ask for a lock it already has

Two-Phase Lock Protocol (2PL)

each transaction considers the following rules:

1. each object used by a transaction has to be locked before the use correspondingly
2. a transaction does not ask for a lock it already has
3. locks are granted based on the compatibility matrix
   in case lock cannot be granted the transaction waits until the lock can be granted

Two-Phase Lock Protocol (2PL)

each transaction considers the following rules:

1. each object used by a transaction has to be locked before the use correspondingly
2. a transaction does not ask for a lock it already has
3. locks are granted based on the compatibility matrix
   in case lock cannot be granted the transaction waits until the lock can be granted
4. after a transaction has released a lock it is not allowed to request a new one

Two-Phase Lock Protocol (2PL)

each transaction considers the following rules:

1. each object used by a transaction has to be locked before the use correspondingly
2. a transaction does not ask for a lock it already has
3. locks are granted based on the compatibility matrix
   in case lock cannot be granted the transaction waits until the lock can be granted
4. after a transaction has released a lock it is not allowed to request a new one
5. at the end of the transaction (EOT) all locks have to be released

Two Phase: Growth and Contraction

important to note:

4. after a transaction has released a lock it is not allowed to request new locks

Two Phase: Growth and Contraction

important to note:

4. after a transaction has released a lock it is not allowed to request new locks

growth phase transaction may request but not release locks

Two Phase: Growth and Contraction

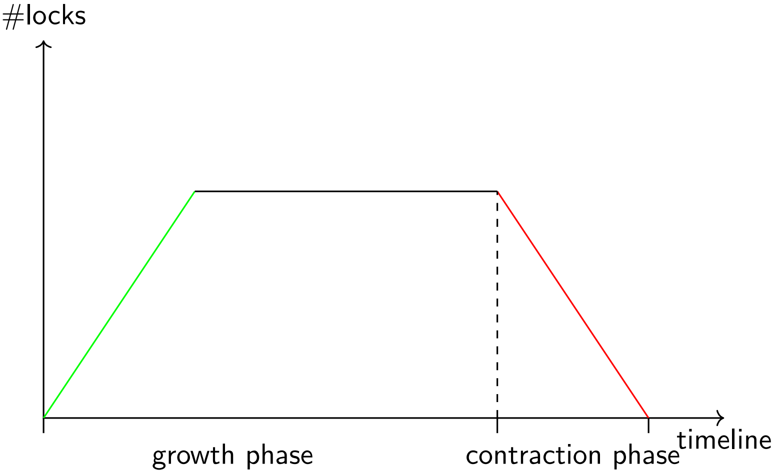
important to note:

4. after a transaction has released a lock it is not allowed to request new locks

growth phase transaction may request but not release locks

contraction phase transaction may release but not request locks

Two Phase: Growth and Contraction



Interleaving of two Transactions as per 2PL

Example (satisfying 2PL)

Schritt	T_1	T_2	remark
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockS**(A)	T_2 has to wait
7.	**lockX**(B)		
8.	read(B)		
9.	**unlockX**(A)		T_2 wake up
10.		read(A)	
11.		**lockS**(B)	T_2 has to wait
12.	write(B)		
13.	**unlockX**(B)		T_2 wake up
14.		read(B)	
15.	**commit**		
16.		**unlockS**(A)	
17.		**unlockS**(B)	
18.		**commit**	

Two Phases Guarantee Conflict Serializability

Example (violation of 2PL)

Schritt	T_1	T_2
1.		
2.	read(A)	
3.	write(A)	
4.		
5.		
6.		
7.		read(A)
8.		write(A)
9.		read(B)
10.		write(B)
9.		
6.		
10.		
11.	read(B)	
12.	write(B)	
13.		

Two Phases Guarantee Conflict Serializability

Example (violation of 2PL)

Schritt	T_1	T_2
1.	**lockX**(A)	
2.	read(A)	
3.	write(A)	
4.	**unlockX**(A)	
5.		**lockX**(A)
6.		**lockX**(B)
7.		read(A)
8.		write(A)
9.		read(B)
10.		write(B)
9.		**unlockX**(A)
6.		**unlockX**(B)
10.	**lockX**(B)	
11.	read(B)	
12.	write(B)	
13.	**unlockX**(B)	

Features of Schedules Based on 2PL

2PL guarantees conflict serializability:

Features of Schedules Based on 2PL

2PL guarantees conflict serializability:

- the order of transactions in a (conflict-) equivalent serial execution is based on the order of the lock requirements for conflicts

Features of Schedules Based on 2PL

2PL guarantees conflict serializability:

- the order of transactions in a (conflict-) equivalent serial execution is based on the order of the lock requirements for conflicts
- inconsistent order of the lock requirements leads to a deadlock

Features of Schedules Based on 2PL

2PL guarantees conflict serializability:

- the order of transactions in a (conflict-) equivalent serial execution is based on the order of the lock requirements for conflicts
- inconsistent order of the lock requirements leads to a deadlock

2PL does not guarantee resettability:

- release of locks before transaction end possible
  $\Rightarrow$ other transaction may access and commit

Not Resettable Schedule at 2PL

Example

step	T_1	T_2	remark
1.			
2.			
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			

Not Resettable Schedule at 2PL

Example

step	T_1	T_2	remark
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.			
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			

Not Resettable Schedule at 2PL

Example

step	T_1	T_2	remark
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 has to wait
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			

Not Resettable Schedule at 2PL

Example

step	T_1	T_2	remark
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 has to wait
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wake up
9.			
10.			
11.			
12.			
13.			
14.			

Not Resettable Schedule at 2PL

Example

step	T_1	T_2	remark
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 has to wait
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wake up
9.	read(B)	read(A)	„T_2 reads from T_1“
10.			
11.			
12.			
13.			
14.			

Not Resettable Schedule at 2PL

Example

step	T_1	T_2	remark
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 has to wait
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wake up
9.	read(B)	read(A)	„ T_2 reads from T_1“
10.		write(A)	
11.		**unlockX**(A)	
12.		**commit**	
13.			
14.			T_2 not resettable!

Not Resettable Schedule at 2PL

Example

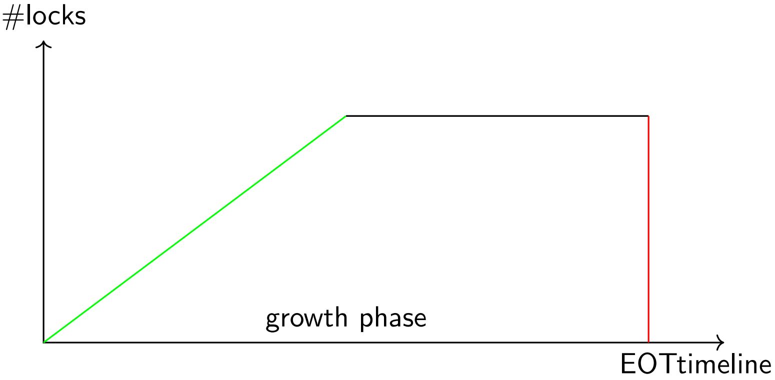
step	T_1	T_2	remark
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 has to wait
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wake up
9.	read(B)	read(A)	„T_2 reads from T_1“
10.		write(A)	
11.		**unlockX**(A)	
12.	write(B)	**commit**	
13.	**unlockX**(B)		
14.	**abort**		T_2 not resettable!

Strict Two-Phase Lock Protocol (Strict 2PL)

- all locks are kept until transaction end

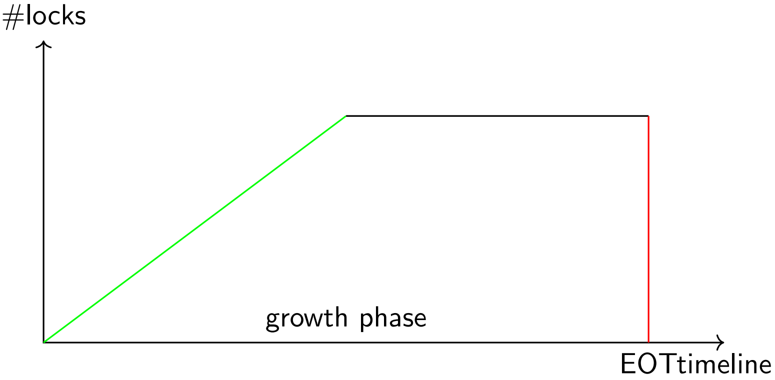
Strict Two-Phase Lock Protocol (Strict 2PL)

- all locks are kept until transaction end



Strict Two-Phase Lock Protocol (Strict 2PL)

- all locks are kept until transaction end
- the strict 2PL-protocol allows only strict schedules



Overview

Deadlocks

Example

step	T_1	T_2	remark
1.	**BOT**		
2.	**lockX**(A)		
3.		**BOT**	
4.		**lockS**(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	**lockX**(B)		T_1 has to wait for T_2
9.		**lockS**(A)	T_2 has to wait for T_1
10.	...	...	$\Rightarrow$ *deadlock*

Deadlock Detection

simple method: time-out strategy

- transaction is reset if it does not make any progress within a fixed time

Deadlock Detection

simple method: time-out strategy

- transaction is reset if it does not make any progress within a fixed time
- problem: “good” choice of time-outs:
    - too small: unnecessary reset
    - too big: deadlocks are detected very late

Deadlock Detection

simple method: time-out strategy

- transaction is reset if it does not make any progress within a fixed time
- problem: "good" choice of time-outs:
    - too small: unnecessary reset
    - too big: deadlocks are detected very late

exact method: wait-for graph

- nodes: active transactions $T_1, T_2, \ldots$
- edges: directed edge $T_i \rightarrow T_j$ if T_i waits for T_j

Deadlock Detection

simple method: time-out strategy

- transaction is reset if it does not make any progress within a fixed time
- problem: “good” choice of time-outs:
  - too small: unnecessary reset
  - too big: deadlocks are detected very late

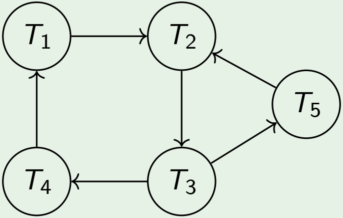
exact method: wait-for graph

- nodes: active transactions $T_1, T_2, \ldots$
- edges: directed edge $T_i \rightarrow T_j$ if T_i waits for T_j
- deadlock: if and only if wait-for graph is cyclic

Wait-For Graph

Example (wait-for graph with two cycles)

1. $T_1 \to T_2 \to T_3 \to T_4 \to T_1$
2. $T_2 \to T_3 \to T_5 \to T_2$



both cycles can be eliminated by resetting T_3

Deadlock Prevention

"preclaiming": transactions are started only if all necessary locks can be granted in the beginning

Deadlock Prevention

"preclaiming": transactions are started only if all necessary locks can be granted in the beginning

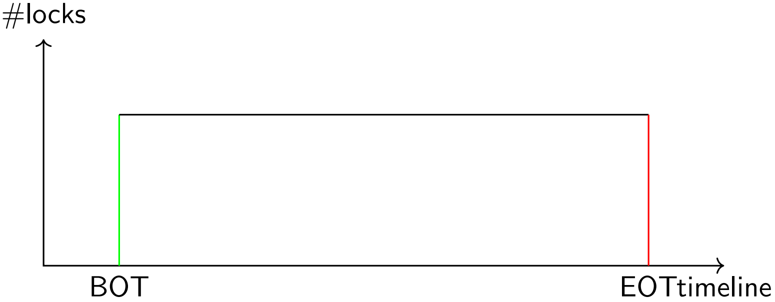
timestamp procedure: each transaction is assigned a timestamp which decides (based on the concrete strategy) whether a transaction has to wait for a lock or not

Conservative 2PL (Preclaiming)

- variant of strict 2PL
- transaction claims all locks it needs in the beginning and keeps them until the end

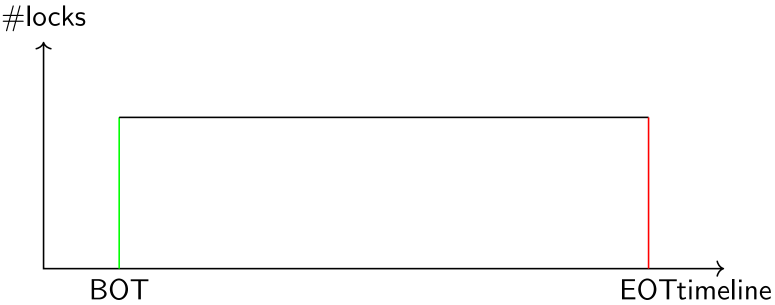
Conservative 2PL (Preclaiming)

- variant of strict 2PL
- transaction claims all locks it needs in the beginning and keeps them until the end



Conservative 2PL (Preclaiming)

- variant of strict 2PL
- transaction claims all locks it needs in the beginning and keeps them until the end



- problem: needed locks are in general not (exactly) known at the beginning of a transaction

Timestamp Procedure – Two Strategies

- each transaction T_i obtains a unique timestamp $TS(T_i)$
- younger transactions obtain higher timestamp

Timestamp Procedure – Two Strategies

- each transaction T_i obtains a unique timestamp $TS(T_i)$
- younger transactions obtain higher timestamp

T_1 wants to obtain a lock that is hold by T_2:

Timestamp Procedure – Two Strategies

- each transaction T_i obtains a unique timestamp $TS(T_i)$
- younger transactions obtain higher timestamp

T_1 wants to obtain a lock that is hold by T_2:

wound-wait strategy older transactions win

Timestamp Procedure – Two Strategies

- each transaction T_i obtains a unique timestamp $TS(T_i)$
- younger transactions obtain higher timestamp

T_1 wants to obtain a lock that is hold by T_2:

wound-wait strategy older transactions win

- T_1 older than T_2: T_2 is aborted

Timestamp Procedure – Two Strategies

- each transaction T_i obtains a unique timestamp $TS(T_i)$
- younger transactions obtain higher timestamp

T_1 wants to obtain a lock that is hold by T_2:

wound-wait strategy older transactions win

- T_1 older than T_2: T_2 is aborted
- otherwise: T_1 waits for the release of the lock via T_2

Timestamp Procedure – Two Strategies

- each transaction T_i obtains a unique timestamp $TS(T_i)$
- younger transactions obtain higher timestamp

T_1 wants to obtain a lock that is hold by T_2:

wound-wait strategy older transactions win

- T_1 older than T_2: T_2 is aborted
- otherwise: T_1 waits for the release of the lock via T_2

wait-die strategy younger transactions are “shy”

Timestamp Procedure – Two Strategies

- each transaction T_i obtains a unique timestamp $TS(T_i)$
- younger transactions obtain higher timestamp

T_1 wants to obtain a lock that is hold by T_2:

wound-wait strategy older transactions win

- T_1 older than T_2: T_2 is aborted
- otherwise: T_1 waits for the release of the lock via T_2

wait-die strategy younger transactions are “shy”

- T_1 older than T_2: T_1 waits for the release of the lock

Timestamp Procedure – Two Strategies

- each transaction T_i obtains a unique timestamp $TS(T_i)$
- younger transactions obtain higher timestamp

T_1 wants to obtain a lock that is hold by T_2:

wound-wait strategy older transactions win

- T_1 older than T_2: T_2 is aborted
- otherwise: T_1 waits for the release of the lock via T_2

wait-die strategy younger transactions are “shy”

- T_1 older than T_2: T_1 waits for the release of the lock
- otherwise: T_1 is aborted

Overview

Hierarchical Granularity Locking

multiple granularity locking (MGL):

- so far: uniform locking granulates
- more efficient: different locking granulates

Hierarchical Granularity Locking

multiple granularity locking (MGL):

- so far: uniform locking granulates
- more efficient: different locking granulates

needed:

- extended locking modes
- extended locking protocol

MGL: Multiple-Granularity Locking

problem: uniform locking granulates (for example: per data set, per table, per page, . . .) might be inefficien; in case granularity is

MGL: Multiple-Granularity Locking

problem: uniform locking granulates (for example: per data set, per table, per page, . . .) might be inefficien; in case granularity is

- too small: slow transactions with lots of data access

MGL: Multiple-Granularity Locking

problem: uniform locking granulates (for example: per data set, per table, per page, . . .) might be inefficien; in case granularity is

- too small: slow transactions with lots of data access
- too big: might block more transactions than actually needed

idea: more flexibility through different locking granulates

- DBMS or user chooses appropriate granularity (data set, table, page, area, data base)

MGL: Multiple-Granularity Locking

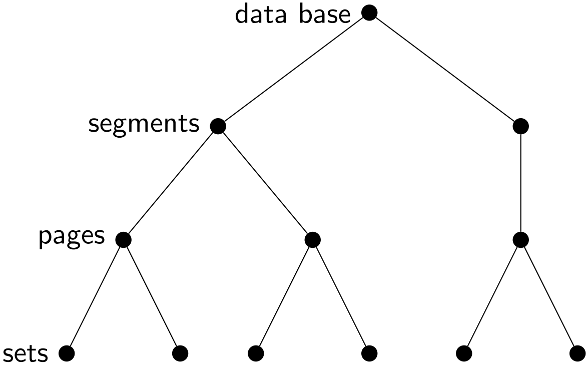
problem: uniform locking granulates (for example: per data set, per table, per page, . . .) might be inefficien; in case granularity is

- too small: slow transactions with lots of data access
- too big: might block more transactions than actually needed

idea: more flexibility through different locking granulates

- DBMS or user chooses appropriate granularity (data set, table, page, area, data base)
- "lock escalation": start with small lock granularity, with increasing amount lock granularity is increased

Hierarchical Ordering of Possible Lock Granularities



segment:
several (logically cohesive) pages (particularly tables)

Extended Locking Modes for MGL

new problem: “overview” over locks with small granularity

- without further measures: before granting locks, all underlying objects have to be checked for locks

Extended Locking Modes for MGL

new problem: "overview" over locks with small granularity

- without further measures: before granting locks, all underlying objects have to be checked for locks

solution: additional locking modes (IS, IX) indicating that there are locks further down the hierarchy

Locking Modes for MGL

locking modes for MGL:

- NL no lock
- S shared lock
- X exclusive lock

Locking Modes for MGL

locking modes for MGL:

- NL no lock
- S shared lock
- X exclusive lock
- IS somewhere lower in the hierarchy there is an intended shared lock (S) (*intention shared lock*)

Locking Modes for MGL

locking modes for MGL:

- NL no lock
- S shared lock
- X exclusive lock
- IS somewhere lower in the hierarchy there is an intended shared lock (S) (*intention shared lock*)
- IX somewhere lower in the hierarchy there is an intended exclusive lock (X) (*intention exclusive lock*)

Compatibility Matrix for MGL

locks		current				
		NL	S	X	IS	IX
requested	S	✓	✓	–	✓	–
	X	✓	–	–	–	–
	IS	✓	✓	–	✓	✓
	IX	✓	–	–	✓	✓

Locking Protocol for MGL

requirement for locks: top-down

Locking Protocol for MGL

requirement for locks: top-down

- before a transaction can lock a node with S or IS it needs for all predecessors in the hierarchy an IS or IX lock

Locking Protocol for MGL

requirement for locks: top-down

- before a transaction can lock a node with S or IS it needs for all predecessors in the hierarchy an IS or IX lock
- before a transaction can lock a node with X or IX it needs for all predecessors an IX lock

Locking Protocol for MGL

requirement for locks: top-down

- before a transaction can lock a node with S or IS it needs for all predecessors in the hierarchy an IS or IX lock
- before a transaction can lock a node with X or IX it needs for all predecessors an IX lock

release of locks: bottom-up

Locking Protocol for MGL

requirement for locks: top-down

- before a transaction can lock a node with S or IS it needs for all predecessors in the hierarchy an IS or IX lock
- before a transaction can lock a node with X or IX it needs for all predecessors an IX lock

release of locks: bottom-up

- an IS or IX lock on a node may only be released when all locks on succeeding nodes have already been released

Database Hierarchy with Locks

Example (MGL protocol)

level:
data base, areas, pages, data sets

3 transactions:

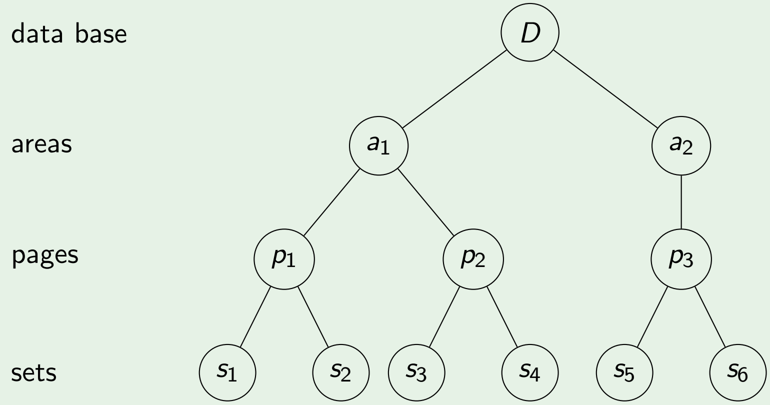
T_1 exclusive lock for pages p_1 below areas a_1

T_2 shared lock for page p_2 below area a_1

T_3 exclusive lock for area a_2

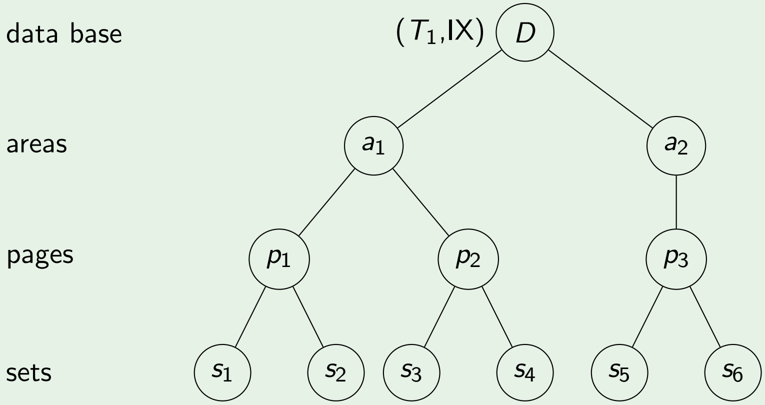
Data Base Hierarchy with Locks

Example (application of the MGL protocol)



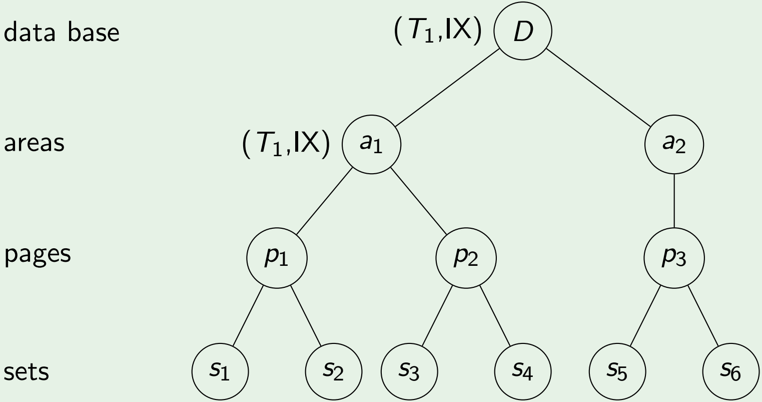
Data Base Hierarchy with Locks

Example (application of the MGL protocol)



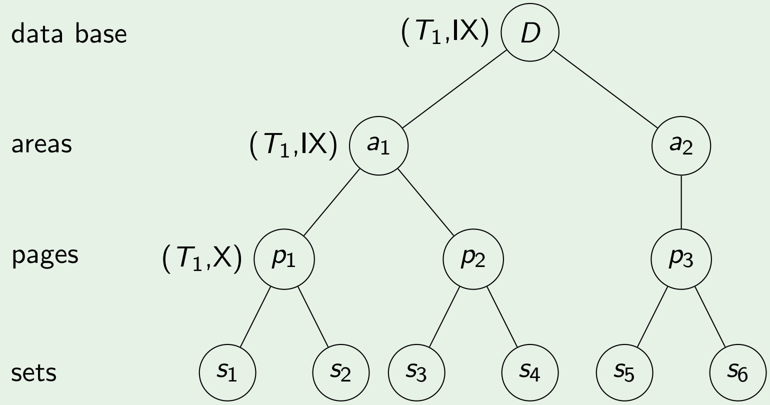
Data Base Hierarchy with Locks

Example (application of the MGL protocol)



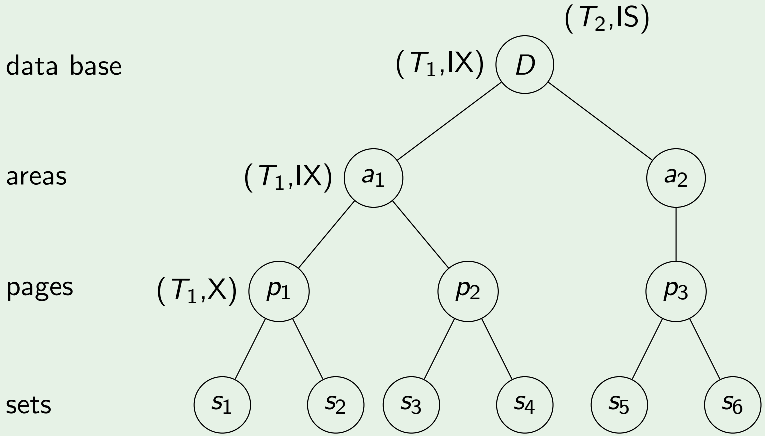
Data Base Hierarchy with Locks

Example (application of the MGL protocol)



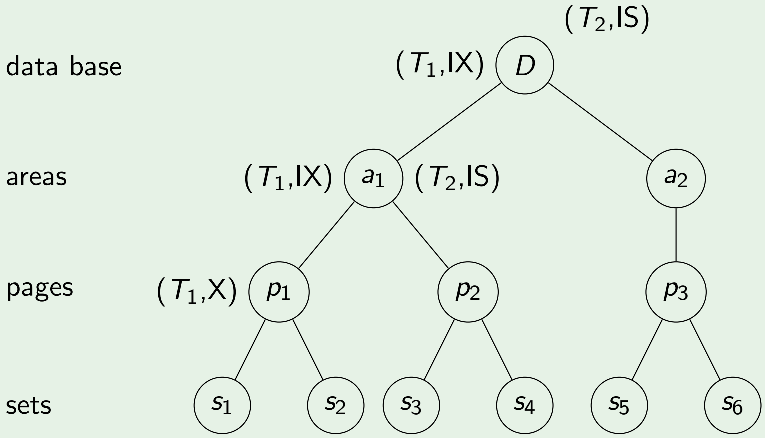
Data Base Hierarchy with Locks

Example (application of the MGL protocol)



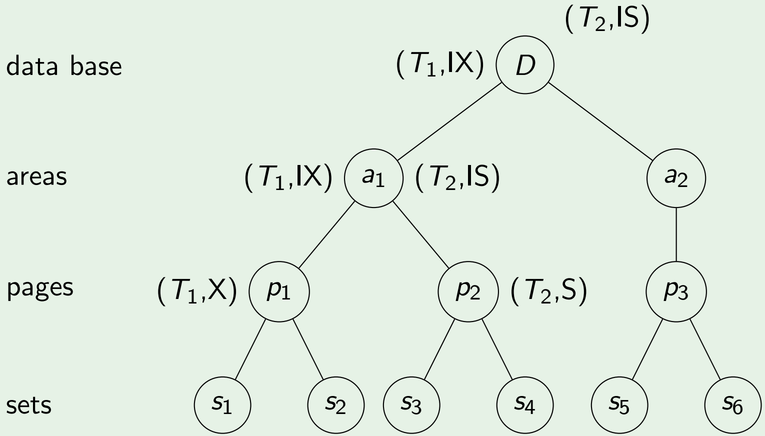
Data Base Hierarchy with Locks

Example (application of the MGL protocol)



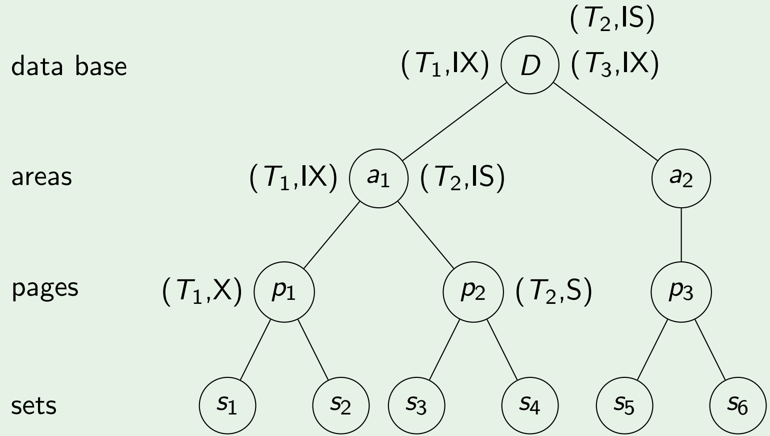
Data Base Hierarchy with Locks

Example (application of the MGL protocol)



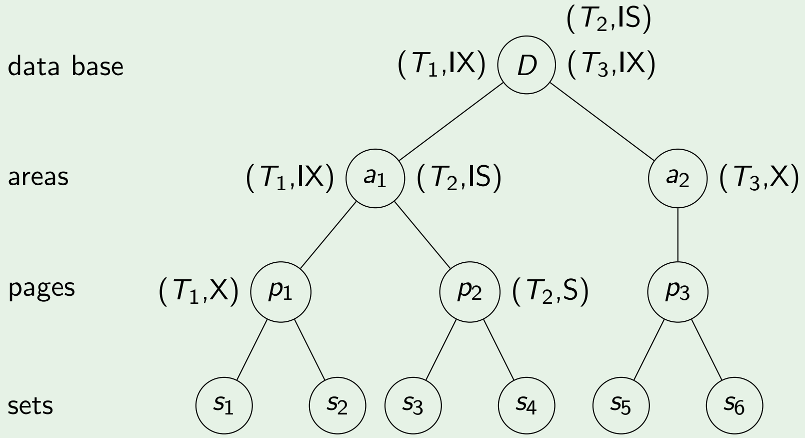
Data Base Hierarchy with Locks

Example (application of the MGL protocol)



Data Base Hierarchy with Locks

Example (application of the MGL protocol)



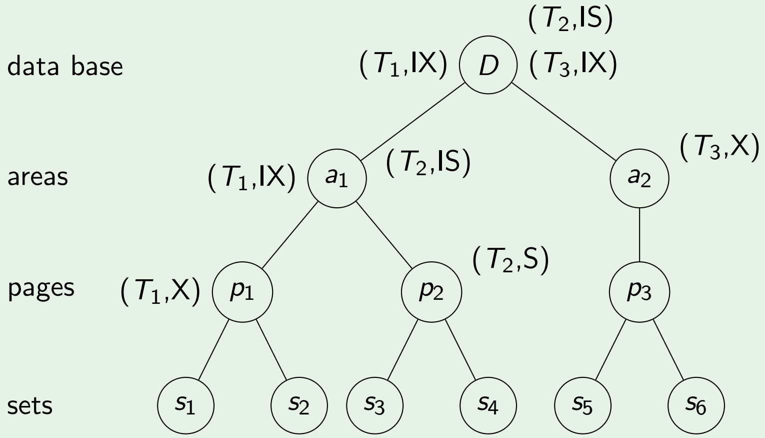
MGL Blocks Transactions

Example (blocking in MGL)

- 3 transactions:
  - T_1 exclusive lock for page p_1 below area a_1
  - T_2 shared lock for page p_2 below area a_1
  - T_3 exclusive lock for area a_2
- continuation:
  - T_4 exclusive lock for set s_3 below page p_2: IX lock request for p_2 fails because of S lock of T_2
  - T_5 shared lock for set s_5 below page p_3 below area a_2: IS lock request for a_2 fails because of X lock of T_3

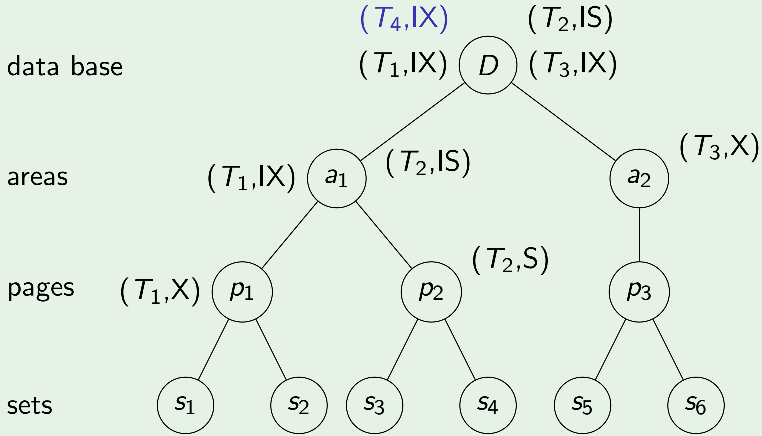
MGL Blocks Transactions

Example (MGL blocks transactions)



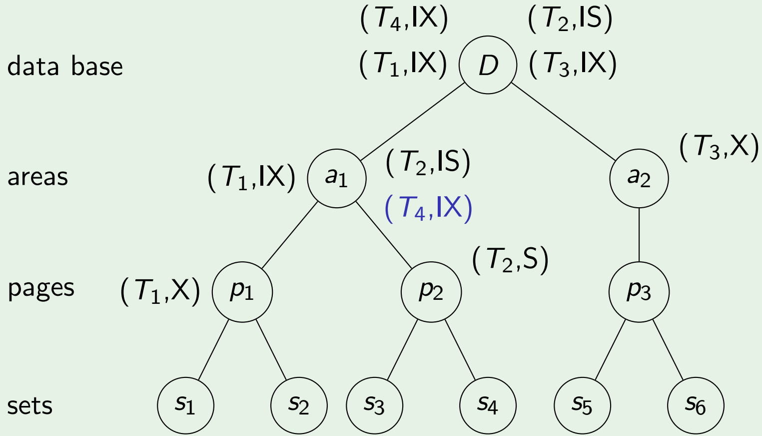
MGL Blocks Transactions

Example (MGL blocks transactions)



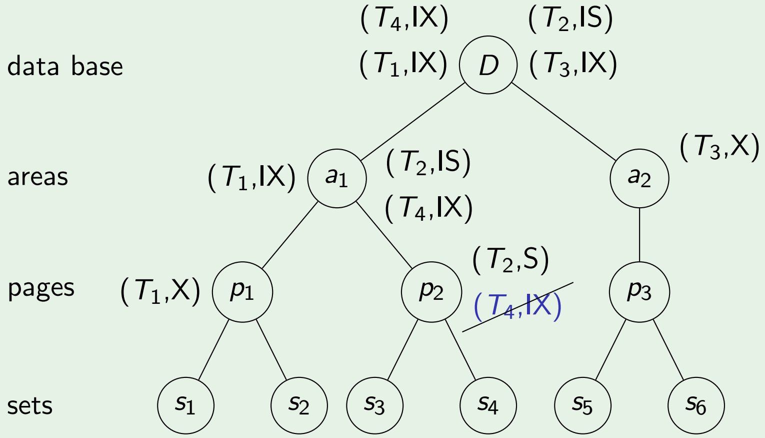
MGL Blocks Transactions

Example (MGL blocks transactions)



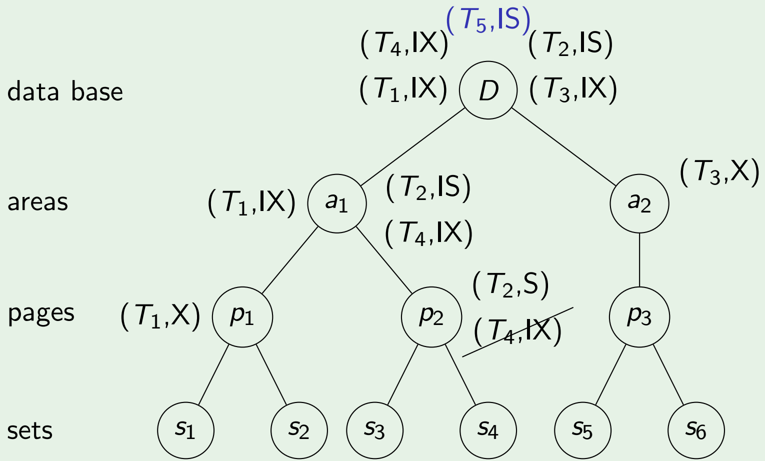
MGL Blocks Transactions

Example (MGL blocks transactions)



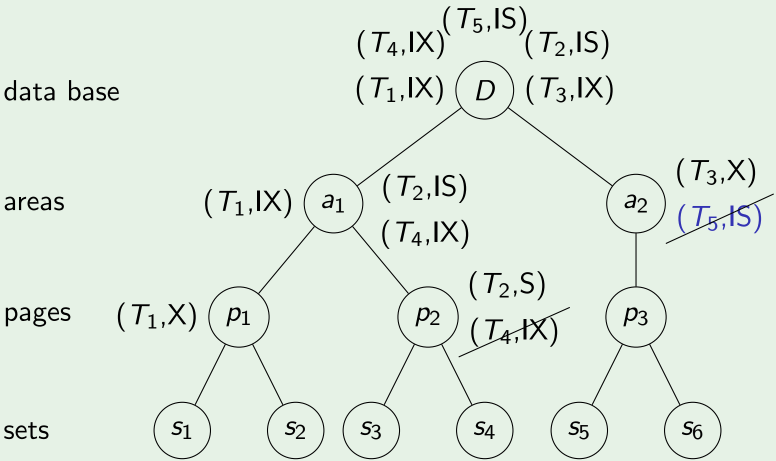
MGL Blocks Transactions

Example (MGL blocks transactions)



MGL Blocks Transactions

Example (MGL blocks transactions)



Overview

Insert/Delete – Operations

so far: assumption that no tuple is inserted or deleted

Insert/Delete – Operations

so far: assumption that no tuple is inserted or deleted

special features of insert and delete operations:

- locking at insert and delete
- phantom problem

Locking at Insert and Delete

delete:

- requires an X lock on the object to be deleted
- other transaction that requires a lock for the object cannot obtain the lock after a `commit`

Locking at Insert and Delete

delete:

- requires an X lock on the object to be deleted
- other transaction that requires a lock for the object cannot obtain the lock after a `commit`

insert:

- transaction obtains an X lock through insert
- avoiding the phantom problem requires additional measures

Phantom Problem

Example (occurrence of the phantom problems)

T_1	T_2
`SELECT COUNT(*)`	
`FROM examine`	
`WHERE grade BETWEEN 1 AND 2`	
	`INSERT INTO examine`
	`VALUES`
	`(29555,5001,2137,1)`
`SELECT COUNT(*)`	
`FROM examine`	
`WHERE grade BETWEEN 1 AND 2`	

Phantom Problem

problem:

Phantom Problem

problem:

- "*if no transaction is aborted, every conflict serializable schedule is serializable*" holds only in case no new entries are inserted

Phantom Problem

problem:

- *"if no transaction is aborted, every conflict serializable schedule is serializable "* holds only in case no new entries are inserted
- (strict) 2PL locks in this case too few data sets:
  - transaction locks only present data sets
  - soundness of 2PL requires locking of all required data sets

Phantom Problem

problem:

- "*if no transaction is aborted, every conflict serializable schedule is serializable* " holds only in case no new entries are inserted
- (strict) 2PL locks in this case too few data sets:
  - transaction locks only present data sets
  - soundness of 2PL requires locking of all required data sets

solution: creation of new relevant data sets has to be prevented

Phantom Problem

problem:

- "*if no transaction is aborted, every conflict serializable schedule is serializable*" holds only in case no new entries are inserted
- (strict) 2PL locks in this case too few data sets:
  - transaction locks only present data sets
  - soundness of 2PL requires locking of all required data sets

solution: creation of new relevant data sets has to be prevented

- table without index: creation of new pages and data sets has to be prevented (for example IS lock on table)
- table with index: additionally the affected index range has to be locked

Overview

Other Synchronisation Methods

Other Synchronisation Methods

- timestamp based synchronisation

Other Synchronisation Methods

- timestamp based synchronisation
- optimistic synchronisation

Timestamp Based Synchronisation

- every transaction T is assigned a unique timestamp $TS(T)$
  - $TS(T_i) < TS(T_j)$ of T_i older than T_j

Timestamp Based Synchronisation

- every transaction T is assigned a unique timestamp $TS(T)$
  - $TS(T_i) < TS(T_j)$ of T_i older than T_j
- each data object A is assigned two values:

Timestamp Based Synchronisation

- every transaction T is assigned a unique timestamp $TS(T)$
  - $TS(T_i) < TS(T_j)$ of T_i older than T_j
- each data object A is assigned two values:
  - `readTS(A)`: TS of the youngest transaction that has read A

Timestamp Based Synchronisation

- every transaction T is assigned a unique timestamp $TS(T)$
  - $TS(T_i) < TS(T_j)$ of T_i older than T_j
- each data object A is assigned two values:
  - `readTS(`A`)`: TS of the youngest transaction that has read A
  - `writeTS(`A`)`: TS of the youngest transaction that has written A

Timestamp Based Synchronisation

- every transaction T is assigned a unique timestamp $TS(T)$
  - $TS(T_i) < TS(T_j)$ of T_i older than T_j
- each data object A is assigned two values:
  - `readTS(`A`)`: TS of the youngest transaction that has read A
  - `writeTS(`A`)`: TS of the youngest transaction that has written A

idea:

- construct serial schedule ordered by the timestamp
  - before the access of T_i to A $TS(T_i)$ is compared with `readTS(`A`)` resp. `writeTS(`A`)` and T_i is reset in case of conflict

Timestamp Based Synch: Read Access

read access $r_i(A)$ (T_i wants to read A):

Timestamp Based Synch: Read Access

read access $r_i(A)$ (T_i wants to read A):

$TS(T_i) < \texttt{writeTS}(A)$: T_i is older than another transaction that has written A

Timestamp Based Synch: Read Access

read access $r_i(A)$ (T_i wants to read A):

$TS(T_i) < \texttt{writeTS}(A)$: T_i is older than another transaction that has written A

- T_i is reset

Timestamp Based Synch: Read Access

read access $r_i(A)$ (T_i wants to read A):

$TS(T_i) < \texttt{writeTS}(A)$: T_i is older than another transaction that has written A

- T_i is reset

$TS(T_i) \geq \texttt{writeTS}(A)$: T_i is younger than another transaction that has written A

Timestamp Based Synch: Read Access

read access $r_i(A)$ (T_i wants to read A):

$TS(T_i) < \texttt{writeTS}(A)$: T_i is older than another transaction that has written A

- T_i is reset

$TS(T_i) \geq \texttt{writeTS}(A)$: T_i is younger than another transaction that has written A

- T_i may read A

Timestamp Based Synch: Read Access

read access $r_i(A)$ (T_i wants to read A):

$TS(T_i) < \texttt{writeTS}(A)$: T_i is older than another transaction that has written A

- T_i is reset

$TS(T_i) \geq \texttt{writeTS}(A)$: T_i is younger than another transaction that has written A

- T_i may read A
- $\texttt{readTS}(A) = \max(TS(T_i), \texttt{readTS}(A))$

Timestamp Based Synch: Write Access

write access $w_i(A)$ (T_i wants to write to A):

Timestamp Based Synch: Write Access

write access $w_i(A)$ (T_i wants to write to A):

$TS(T_i) < \texttt{readTS}(A)$: a younger transaction has read A already but should have read the value T_i wants to write to A

Timestamp Based Synch: Write Access

write access $w_i(A)$ (T_i wants to write to A):

$TS(T_i) < \texttt{readTS}(A)$: a younger transaction has read A already but should have read the value T_i wants to write to A

- T_i is reset

Timestamp Based Synch: Write Access

write access $w_i(A)$ (T_i wants to write to A):

$TS(T_i) < \texttt{readTS}(A)$: a younger transaction has read A already but should have read the value T_i wants to write to A

- T_i is reset

$TS(T_i) < \texttt{writeTS}(A)$: a younger transaction has written to A and is overwritten by T_i

Timestamp Based Synch: Write Access

write access $w_i(A)$ (T_i wants to write to A):

$TS(T_i) < \texttt{readTS}(A)$: a younger transaction has read A already but should have read the value T_i wants to write to A

- T_i is reset

$TS(T_i) < \texttt{writeTS}(A)$: a younger transaction has written to A and is overwritten by T_i

- T_i is reset

Timestamp Based Synch: Write Access

write access $w_i(A)$ (T_i wants to write to A):

$TS(T_i) < \texttt{readTS}(A)$: a younger transaction has read A already but should have read the value T_i wants to write to A

- T_i is reset

$TS(T_i) < \texttt{writeTS}(A)$: a younger transaction has written to A and is overwritten by T_i

- T_i is reset

else:

- T_i may write to A
- $\texttt{writeTS}(A) = TS(T_i)$

Timestamp Based Sync: Example

Example

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

Timestamp Based Sync: Example

Example

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0
4			

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0
4	$r_1(A)$		

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0
4	$r_1(A)$	1	0

Timestamp Based Sync: Example

Example

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5			

Timestamp Based Sync: Example

Example

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$		

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \to \mathtt{BOT}_2 \to \mathtt{BOT}_3 \to r_1(A) \to w_2(A) \to r_3(A) \to r_1(A) \to w_3(A) \to w_2(A) \to c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6			

FAKULTÄT FÜR INFORMATIK
Faculty of Informatics

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \to \mathtt{BOT}_2 \to \mathtt{BOT}_3 \to r_1(A) \to w_2(A) \to r_3(A) \to r_1(A) \to w_3(A) \to w_2(A) \to c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2
7			

Timestamp Based Sync: Example

Example

$\texttt{BOT}_1 \to \texttt{BOT}_2 \to \texttt{BOT}_3 \to r_1(A) \to w_2(A) \to r_3(A) \to r_1(A) \to w_3(A) \to w_2(A) \to c_2$

TS	action	readTS(A)	writeTS(A)
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2
7	$\texttt{reset}_1$ $[r_1(A)]$	3	2

Timestamp Based Sync: Example

Example

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\mathtt{BOT}_1$	0	0
2	$\mathtt{BOT}_2$	0	0
3	$\mathtt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2
7	$\mathtt{reset}_1$ $[r_1(A)]$	3	2
8	$w_3(A)$	3	3

Timestamp Based Sync: Example

Example

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A)	writeTS(A)
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2
7	$\texttt{reset}_1$ $[r_1(A)]$	3	2
8	$w_3(A)$	3	3
9	$\texttt{reset}_2$ $[w_2(A)]$	3	3

Timestamp Based Synch: Features

- deadlocks do not occur
- yields conflict serializable schedules
- order in serial schedule corresponds to timestamps

Timestamp Based Synch: Features

- deadlocks do not occur
- yields conflict serializable schedules
- order in serial schedule corresponds to timestamps
- **but:** schedules might not be resettable

Timestamp Based Synch: Features

- deadlocks do not occur
- yields conflict serializable schedules
- order in serial schedule corresponds to timestamps
- **but:** schedules might not be resettable

solution:

Timestamp Based Synch: Features

- deadlocks do not occur
- yields conflict serializable schedules
- order in serial schedule corresponds to timestamps
- **but:** schedules might not be resettable

solution:

resettable: delay the `commit` until all transactions to be read have been completed (maintain a list)

Timestamp Based Synch: Features

- deadlocks do not occur
- yields conflict serializable schedules
- order in serial schedule corresponds to timestamps
- **but:** schedules might not be resettable

solution:

resettable: delay the `commit` until all transactions to be read have been completed (maintain a list)

strict: apply all writing procedures at the end of the transaction atomically

Timestamp Based Synch: Features

- deadlocks do not occur
- yields conflict serializable schedules
- order in serial schedule corresponds to timestamps
- **but:** schedules might not be resettable

solution:

resettable: delay the `commit` until all transactions to be read have been completed (maintain a list)

strict: apply all writing procedures at the end of the transaction atomically

alternatively: mark data not yet set and delay access

Timestamp Based Synch: Features

- deadlocks do not occur
- yields conflict serializable schedules
- order in serial schedule corresponds to timestamps
- **but:** schedules might not be resettable

solution:

resettable: delay the `commit` until all transactions to be read have been completed (maintain a list)

strict: apply all writing procedures at the end of the transaction atomically

alternatively: mark data not yet set and delay access

Strict Schedules Through Timestamps

procedure:

- *"dirty bit":* mark data sets modified by a still active transaction
- in case dirty bit is set all other transactions that want to access the data will be delayed
  - only reading access: avoids cascading reset
  - write and read access: guarantees strict schedules

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	$\mathtt{readTS}(A)$:	$\mathtt{writeTS}(A)$	$d(A)$

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A):	writeTS(A)	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	$\mathtt{readTS}(A)$:	$\mathtt{writeTS}(A)$	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \to \mathtt{BOT}_2 \to \mathtt{BOT}_3 \to r_1(A) \to w_2(A) \to r_3(A) \to r_1(A) \to w_3(A) \to w_2(A) \to c_2$

TS	action	readTS(A):	writeTS(A)	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	$\mathtt{readTS}(A)$:	$\mathtt{writeTS}(A)$	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6				

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \to \mathtt{BOT}_2 \to \mathtt{BOT}_3 \to r_1(A) \to w_2(A) \to r_3(A) \to r_1(A) \to w_3(A) \to w_2(A) \to c_2$

TS	action	`readTS`(A):	`writeTS`(A)	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$r_3(A)$			

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A):	writeTS(A)	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\mathtt{block}_3\ [r_3(A)]$	1	2	✗

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	$\mathtt{readTS}(A)$:	$\mathtt{writeTS}(A)$	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\mathtt{block}_3\ [r_3(A)]$	1	2	✗
7				

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	$\mathtt{readTS}(A)$:	$\mathtt{writeTS}(A)$	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\mathtt{block}_3$ $[r_3(A)]$	1	2	✗
7	$\mathtt{reset}_1$ $[r_1(A)]$	1	2	✗

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	$\mathtt{readTS}(A)$:	$\mathtt{writeTS}(A)$	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\mathtt{block}_3\ [r_3(A)]$	1	2	✗
7	$\mathtt{reset}_1\ [r_1(A)]$	1	2	✗
8	$w_2(A)$	1	2	✗

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	$\mathtt{readTS}(A)$:	$\mathtt{writeTS}(A)$	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\mathtt{block}_3\ [r_3(A)]$	1	2	✗
7	$\mathtt{reset}_1\ [r_1(A)]$	1	2	✗
8	$w_2(A)$	1	2	✗
9	c_2	1	2	

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A):	writeTS(A)	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\mathtt{block}_3\ [r_3(A)]$	1	2	✗
7	$\mathtt{reset}_1\ [r_1(A)]$	1	2	✗
8	$w_2(A)$	1	2	✗
9	c_2	1	2	
10	$r_3(A)$	3	2	

Strict Schedule Through Timestamp and Dirty Bit

$\mathtt{BOT}_1 \rightarrow \mathtt{BOT}_2 \rightarrow \mathtt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	action	readTS(A):	writeTS(A)	$d(A)$
1	$\mathtt{BOT}_1$	0	0	
2	$\mathtt{BOT}_2$	0	0	
3	$\mathtt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\mathtt{block}_3$ $[r_3(A)]$	1	2	✗
7	$\mathtt{reset}_1$ $[r_1(A)]$	1	2	✗
8	$w_2(A)$	1	2	✗
9	c_2	1	2	
10	$r_3(A)$	3	2	
11	$w_3(A)$	3	3	✗

FAKULTÄT FÜR INFORMATIK Faculty of Informatics

Strict Schedules Through Timestamp

`commit` of T_i:

- update dirty bits

Strict Schedules Through Timestamp

`commit` of T_i:

- update dirty bits

`roll-back` of T_i:

- for every data set A written by the transaction:
  reset `writeTS`(A) to the value before the write access of T_i
- update dirty bits
- *remark:* the last transaction written before T_i was completed successfully

Optimistic Synchronisation

general idea:

- execute transactions
- check actions and decide whether conflicts have occurred or not
- in case of a problem: reset transaction

Optimistic Synchronisation

general idea:

- execute transactions
- check actions and decide whether conflicts have occurred or not
- in case of a problem: reset transaction

in the following: a concrete method

Optimistic Synchronisation

transaction split in 3 stages:

1. reading stage
   - all operations of the transaction are executed
   - but: writing operations only on local copies
2. validation stage
   - check whether the transaction may be commuted
   - conflicts are recognized through timestamps (obtained ordered by entrance into the validation stage)
3. writing stage
   - after successful validation modifications are applied to the data base

Optimistic Synchronisation: Validation Stage

validation of T_i:

for all transactions T_a with $TS(T_a) < TS(T_i)$ (attention: TS is assigned after entering the validation stage!) one of the two conditions has to be satisfied:

1. T_a was completed already when T_i started
2. $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$

Optimistic Synchronisation: Validation Stage

validation of T_i:
for all transactions T_a with $TS(T_a) < TS(T_i)$ (attention: TS is assigned after entering the validation stage!) one of the two conditions has to be satisfied:

1. T_a was completed already when T_i started
2. $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$
   - $WriteSet(T_a)$ all data objects written by T_a

Optimistic Synchronisation: Validation Stage

validation of T_i:

for all transactions T_a with $TS(T_a) < TS(T_i)$ (attention: TS is assigned after entering the validation stage!) one of the two conditions has to be satisfied:

1. T_a was completed already when T_i started
2. $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$
   - $WriteSet(T_a)$ all data objects written by T_a
   - $ReadSet(T_i)$ all data objects read by T_i

Optimistic Synchronisation: Validation Stage

validation of T_i:
for all transactions T_a with $TS(T_a) < TS(T_i)$ (attention: TS is assigned after entering the validation stage!) one of the two conditions has to be satisfied:

1. T_a was completed already when T_i started
2. $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$
    - $WriteSet(T_a)$ all data objects written by T_a
    - $ReadSet(T_i)$ all data objects read by T_i

important for soundness:
validation and writing stage have to be atomic (only one transaction is allowed per stage 2 and 3)

Overview

Transaction Management in SQL

```
SET TRANSACTION
  [READ WRITE | READ ONLY]
  [ISOLATION LEVEL { READ UNCOMMITED |
                     READ COMMITED |
                     REPEATABLE READ |
                     SERIALIZABLE }]
```

Transaction Management in SQL

```
SET TRANSACTION
  [READ WRITE | READ ONLY]
  [ISOLATION LEVEL { READ UNCOMMITED |
                     READ COMMITED |
                     REPEATABLE READ |
                     SERIALIZABLE }]
```

access mode:

- read only: only reading access ($\Rightarrow$ only reading locks)
- read write: default

Transaction Management in SQL

```
SET TRANSACTION
  [READ WRITE | READ ONLY]
  [ISOLATION LEVEL { READ UNCOMMITED |
                     READ COMMITED |
                     REPEATABLE READ |
                     SERIALIZABLE }]
```

access mode:

- read only: only reading access ($\Rightarrow$ only reading locks)
- read write: default

isolation level:

- offer a variety of isolation grades, allow to increase parallelizability

Isolation Levels in SQL

`READ UNCOMMITED` weakest stage; can read modifications that have not been set yet and therefore inconsistent data base stages; allowed only for `READ ONLY` transactions

Isolation Levels in SQL

`READ UNCOMMITED` weakest stage; can read modifications that have not been set yet and therefore inconsistent data base stages; allowed only for `READ ONLY` transactions

`READ COMMITED` each operation only sees data sets that were committed already before the start of the operation

Isolation Levels in SQL

`READ UNCOMMITED` weakest stage; can read modifications that have not been set yet and therefore inconsistent data base stages; allowed only for `READ ONLY` transactions

`READ COMMITED` each operation only sees data sets that were committed already before the start of the operation

`REPEATABLE READ` all operations of the transaction only see datasets that were committed before the first action of the transaction

Isolation Levels in SQL

`READ UNCOMMITED` weakest stage; can read modifications that have not been set yet and therefore inconsistent data base stages; allowed only for `READ ONLY` transactions

`READ COMMITED` each operation only sees data sets that were committed already before the start of the operation

`REPEATABLE READ` all operations of the transaction only see datasets that were committed before the first action of the transaction

`SERIALIZABLE` highest stage

Isolation Levels and Possible Anomalies

isolation level	dirty read	unrepeatable read	phantom problem
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	–	possible	possible
REPEATABLE READ	–	–	possible
SERIALIZABLE	–	–	–

Summary

Protocols and Their Most Important Features

protocol	equivalent serial schedule	further features	deadlock possible
2PL	order of the locking requirements at conflicts	in general not resettable	yes
strict 2PL	as 2PL	strikt	yes
strict 2PL + deadlock avoidance	as 2PL	strict	no
timestamp based	time of BOT	strict variant exists	no
optimistic	time of validation	strict	no