Chapter 6
Running a Database: Safety and Correctness Issues

- Transactions
- Safety against failure

Not discussed here:

- Access control, Authentication

6.1 Transactions: Properties and Basic Notions

Transaction:

- a unit of work from the user's point of view.
- for the DBS: a process, characterized by a sequence of database accesses
- requirements: **ACID-properties**:

Atomicity: A transaction is (logically) a unit that cannot be further decomposed: its effect is *atomic*, i.e., all updates are executed completely, or nothing at all ("all-or-nothing").

Concistency: A transaction is a correct transition from one state to another. The final state is not allowed to violate any integrity condition (otherwise the transaction is undone and rejected).

Isolation: Databases are multi-user systems. Although transactions are running *concurrently*, this is hidden against the user
(i.e., after starting a transaction, the user does not see changes by other transactions until finishing his transaction, *simulated* single-user).

Durability: If a transaction completes successfully, all its effects are durable (=persistent). I.e., no error situation (including system crash!) is allowed to undo them $\Rightarrow$ safety.

Transactions consist of elementary actions:

- Read access: READ
  By READ A (RA), the value of a DB-object A from the DB is copied to the local workspace of the transaction.
- Write Access: WRITE
  By WRITE A (WA), the value of a DB-object A is copied from the local workspace of the transaction to the DB.
- BEGIN WORK and COMMIT WORK denote its begin (BOT - begin of transaction) and its successful completion (EOT - end of transaction).

$\Rightarrow$ of the form BOT RA RB RC ... WA ... RD ... WE EOT

- ROLLBACK WORK for undoing all its effects (ABORT).
- These elementary actions are *physically* atomic. At every timepoint, only one such action is executed.
- in contrast, several transactions may be **interleaved** (see below).

6.2 Transaction models

FLAT TRANSACTIONS

Basic transaction model: Transactions are a "flat" (and short) sequence of elementary actions without additional structure.

Example 6.1

Outline of a simple transaction for transferring money from account A to account B:

1. *BEGIN WORK*
3. *debit (READ and WRITE) money from account A.*
4. *book money (READ and WRITE) on account B.*
5. *if account A negative, then ROLLBACK, otherwise COMMIT WORK.* □

FLAT TRANSACTIONS WITH SAVEPOINTS

Limits of simple flat transactions: long transactions, e.g., travel booking (hotel, several flights, rental car)

- partial rollback, for trying alternative continuations:
- SAVE WORK defines savepoints (intermediate states)
- sequences between savepoints are *atomic* (but in general not consistent and durable)
- ROLLBACK WORK(i) undoes effects back to savepoint i
- COMMIT WORK commits the whole transaction (ACID)
- complete ROLLBACK WORK undoes the whole transaction

Other Transaction Models

- Nested transactions

6.3 Multi-User Aspects

- In general, at any timepoint, several transactions are running.
- means **interleaving** (i.e., one step here, one step there, and again one step here)
- not necessarily true **parallelism** (requires multi-processor systems)
- techniques for interleaving also sufficient for parallelism

Goal of multi-user policies: allow for as much interleaving as possible without the risk of "unintended" results

Problem: transactions run on **shared** data.
$\Rightarrow$ enforce virtual isolation

TYPICAL ERROR SITUATIONS

Lost update

population update	population update
SELECT population INTO pop	
FROM City WHERE name=“Berlin”	
	SELECT population INTO pop
	FROM City WHERE name=Berlin
pop := pop + 2000;	
	pop := pop + 1000;
UPDATE City	
SET population = pop	
WHERE Name = “Berlin”	
	UPDATE City
	SET population = pop
	WHERE Name = “Berlin”

- the first update is lost

Dirty Read

population update	population update
UPDATE City	
SET population = population + 1000	
WHERE Name = “Berlin”	
	SELECT population INTO pop
	FROM City
	WHERE Name = “Berlin”
⋮	pop := pop * 1.05;
	UPDATE City
	SET population = pop
	WHERE Name = “Berlin”
ABORT (i.e., ROLLBACK)	
(what must be done now?)	

- The second transaction reads and uses a value that has been undone

Non-repeatable Read

Sum of Population	Change Population
SELECT population INTO pop	
FROM City	
WHERE Name = “Berlin”	
sum := sum + pop	
	UPDATE City
	SET population= population + 1000
	WHERE Name = “Berlin”
	UPDATE City
	SET population= population + 2000
	WHERE Name = “Munich”
SELECT population INTO pop	
FROM City	
WHERE Name = “Munich”	
sum := sum + population	

- The value computed in sum does not correspond to any database state

Phantom

- sum of population of provinces equals population of country

Check sum of population	Insert new province
SELECT SUM(population) INTO sum	/* people say that in the summer, the holiday island
FROM Province	Mallorca (Spain) becomes a german province */
WHERE Country = “D”	
	INSERT INTO Province (name, country, population)
	VALUES (“Mallorca”, “D”, 100000)
	UPDATE Country
	SET population = population + 100000
	WHERE Code = “D”
SELECT population INTO countrypop	
FROM Country WHERE Code = “D”	
IF countrypop $\neq$ sum THEN	
<error handling>	

- similar to non-repeatable read

6.4 Serializability

- A **schedule** wrt. a set of transactions is an interleaving of their (elementary) actions that does not change the inner order of each of the transactions.
- A schedule is **serial** if the actions of each individual transaction are immediately following each other (no interleaving).

Example 6.2 (Bank Accounts: Transferring Money from A to B)
T_1 = RA; A:=A-10; WA; RB; B:=B+10; WB, *T_2 = RA; A:=A-20; WA; RB; B:=B+20; WB*

Some schedules (without computation steps):

$$S_1 = R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B \quad \textit{(serial)}$$
$$S_2 = R_1A\ R_2A\ W_1A\ W_2A\ R_1B\ R_2B\ W_1B\ W_2B$$
$$S_3 = R_1A\ W_1A\ R_2A\ W_2A\ R_1B\ W_1B\ R_2B\ W_2B$$
$$S_4 = R_1A\ W_1A\ R_2A\ W_2A\ R_2B\ W_2B\ R_1B\ W_1B$$
$$S_5 = R_2A\ W_2A\ R_2B\ W_2B\ R_1A\ W_1A\ R_1B\ W_1B \quad \textit{(serial)}$$

□

... which of them are “good”?

Serializability Criterion for Parallel Transactions

- “Isolation” requirement
  Transaction may not see results from other (not yet committed) ones
- The serial ones are good.
- are there other “good” ones?

Definition 6.1
*A schedule is **serializable** if and only if there exists an equivalent serial schedule.* □

Example 6.2 (Cont'd: Bank Accounts Interleaved)

A=B=10; T_1*: RA; A:=A-10; WA; RB; B:=B+10; WB* T_2*: RA; A:=A-20; WA; RB; B:=B+20; WB*

T_1	T_2	T_1	T_2	T_1	T_2	T_1	T_2
RA		RA		RA		RA	
A:=A-10			RA	A:=A-10		A:=A-10	
WA		A:=A-10		WA		WA	
RB			A:=A-20		RA		RA
B:=B+10		WA			A:=A-20		A:=A-20
WB			WA		WA		WA
	RA	RB		RB			RB
	A:=A-20		RB	B:=B+10			B:=B+20
	WA	B:=B+10		WB			WB
	RB		B:=B+20		RB	RB	
	B:=B+20	WB			B:=B+20	B:=B+10	
	WB		WB		WB	WB	
serial		*not serializable*		*serializable*		*serializable?*	
A=-20, B=40		*A=-10, B=30*		*A=-20,B=40*		*A=-20,B=40*	
A+B=20		*A+B=20*		*A+B=20*		*A+B=20*	

□

Problem: what means "equivalence" in this context?

- consider each step in each transaction?
  Then, (4) is not equivalent with (1):
  in (1) T_1 reads $B = 10$, in (4), T_1 reads $B = 30$
- consider the initial and final database state?
  Then, (4) and (1) would be equivalent.

Exercise 6.1

Consider again Example 6.2 for A=B=10;
T_1*: RA; A:=A/2; WA; RB; B:=B/2; WB (Transforming DM to Euro) and*
T_2*: RA; A:=A-10; WA; RB; B:=B+10; WB (Money Transfer).*
Consider also the schedule T_2T_1*.*

□

6.4.1 Formalization of the Semantics of Transactions

- Theory & algorithms depend only on the READ and WRITE actions, not on the semantics of the computations in-between.

  Transactions T and schedules S are represented as a sequence of their READ- and WRITE-Actions (actions are assigned to transactions by indexing).

Let $\mathcal{D}$ the domain of the database objects.

Let WA a write action of a transaction T and $RB_1, \ldots, RB_k$ $k \geq 0$ the read actions that are executed by T before WA.

The value written to A by WA is denoted by

$$f_{T,A}(B_1, \ldots, B_k),$$

where

$$f_{T,A} : \mathcal{D}^k \to \mathcal{D}$$

is called the **Interpretation** of the write action WA in T

($f_{T,A}$ encodes the functional relationship (computation) between the read-values and the written value)

Example 6.2 (Cont'd)

Consider again the transactions $T_1 = RA\ WA\ RB\ WB$ *and* $T_2 = RA\ WA\ RB\ WB$

Let the initial state be given by values A_0, B_0.

Schedule $T_1 T_2$ *(serial)*			*Schedule* $T_2 T_1$ *(serial)*		
$T_1:$	RA	A_0	$T_2:$	RA	A_0
	WA	$f_{T_1,A}(A_0)$		WA	$f_{T_2,A}(A_0)$
	RB	B_0		RB	B_0
	WB	$f_{T_1,B}(A_0, B_0)$		WB	$f_{T_2,B}(A_0, B_0)$
$T_2:$	RA	$f_{T_1,A}(A_0)$	$T_1:$	RA	$f_{T_2,A}(A_0)$
	WA	$f_{T_2,A}(f_{T_1,A}(A_0))$		WA	$f_{T_1,A}(f_{T_2,A}(A_0))$
	RB	$f_{T_1,B}(A_0, B_0)$		RB	$f_{T_2,B}(A_0, B_0)$
	WB	$f_{T_2,B}(f_{T_1,A}(A_0), f_{T_1,B}(A_0, B_0))$		WB	$f_{T_1,B}(f_{T_2,A}(A_0), f_{T_2,B}(A_0, B_0))$

□

Definition 6.2
Two schedules S, S' (of the same set of transactions) are equivalent, if for every initial state and every interpretation of the writing actions,

(1) all transactions read the same values in S and S', and

(2) S and S' generate the same final states □

Exercise 6.2
Consider again Example 6.2.

Show by the detailed tables with $f(...)$ that Schedule S_2 and Schedule S_4 are not serializable, but Schedule S_3 is serializable.

Give at least one more serializable schedule. □

6.4.2 Theoretical Investigations

Consider a schedule S together with two additional distinguished transactions T_0, T_∞:
T_0 generates the initial state, and T_∞ reads the final state of S.

- T_0 is a transaction that executes a write action for every database object for which S executes a read or write action.
- T_∞ is a transaction that executes a read action for every database object for which S executes a read or write action.

The schedule $\hat{S} = T_0\ S\ T_\infty$ is the **augmented schedule** to S.

Assumption (without loss of generality):

- each transaction reads and writes an object at most once,
- if a transaction reads *and* writes an object, then reading happens before writing.

Corollary 6.1
Two schedules S, S' (of the same set of transactions) are equivalent if and only if for every interpretation of the write actions, all transactions read the same values for $\hat{S}$ and $\hat{S}'$. □

DEPENDENCY GRAPHS

Consider a schedule S. The **D-Graph** (*dependency graph*) of S is a directed graph $DG(S) = (V, E)$ where V is the set of actions in $\hat{S}$ and E is the set of edges given as follows ($i \neq j$):

- if $\hat{S} = \ldots R_i B \ldots W_i A \ldots$, then $R_i B \to W_i A \in E$,
  (i.e., T_i reads B (and possible uses it) and then writes a value A)
- if $\hat{S} = \ldots W_i A \ldots R_j A \ldots$, then $W_i A \to R_j A \in E$,
  if there is no write action to A between $W_i A$ and $R_j A$ in $\hat{S}$.
  (i.e., T_j reads a value A that has been written by T_i)

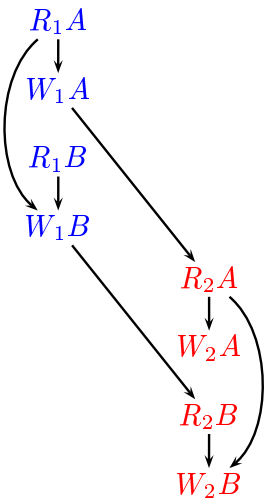
A transaction T' is **dependent** of a transaction T, if in S, either T' reads a value that has been written by T, or by a transaction that is dependent on T.

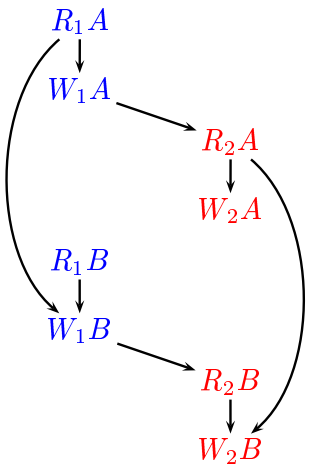
Example 6.3

Consider again Example 6.2: $T_1 = RA\ WA\ RB\ WB$; $T_2 = RA\ WA\ RB\ WB$

Consider the serial schedule $T_1 T_2$ *and* $S_3 = R_1 A\ W_1 A\ R_2 A\ W_2 A\ R_1 B\ W_1 B\ R_2 B\ W_2 B$.

Dependency graphs:





□

Theorem 6.1

Two schedules S, S' *(of the same transactions) are equivalent if and only if* $DG(S) = DG(S')$.

□

CONFLICT GRAPHS

Consider a schedule S. The **C-Graph** (*conflict graph*) of S is a directed graph $CG(S) = (V, E)$ where V is the set of Transactions in $\hat{S}$ and E is a set of edges give as follows ($i \neq j$):

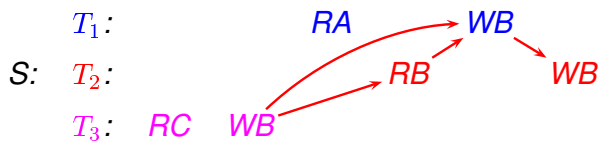
- if $S = \ldots W_i A \ldots R_j A \ldots$ then $T_i \to T_j \in E$, if there is no write action to A between $W_i A$ and $R_j A$ in S **(WR-conflict)**.
- if $S = \ldots W_i A \ldots W_j A \ldots$ then $T_i \to T_j \in E$, if there is no write action to A between $W_i A$ and $W_j A$ in S **(WW-conflict)**.
- if $S = \ldots R_i A \ldots W_j A \ldots$ then $T_i \to T_j \in E$, if there is no write action to A between $R_i A$ and $W_j A$ in S **(RW-conflict)**.

Theorem 6.2
A schedule S is serializable, if $CG(S)$ is cycle-free. □

Proof Sketch: Since $CG(S)$ is cycle-free, there is a topological order of the nodes (i.e., of the transactions).
Let S' the serial schedule according to this ordering. Then, $DG(S) = DG(S')$.

Note that there are serializable schedules whose C-Graph contains cycles:

Example 6.4
Consider the following schedule S:



The C-Graph containing the edges (3,1), (3,2), (2,1), and (1,2) is cyclic.

Nevertheless, $S' = T_1 T_3 T_2$ is an equivalent serial schedule.

Why this? $W_1 B$ is not used anywhere in S. It is also not used in S'. □

Definition 6.3
*A schedule is **C-serializable** (conflict-serializable) if its C-Graph is cycle-free.* □

Note: In the sequel, serializability always means C-serializability.

6.4.3 Scheduler

The **Scheduler** of a database system ensures that only serializable schedules are executed. This can be done by different **strategies**.

Input: a set of actions of a set of transactions (to be executed)
Output: a serializable sequence (= the schedule to be actually executed) of these actions

Strategies

- **Based on the conflict graph:**
  Let S the current (partial) schedule and $action$ the next action of some transaction T.
  If $CG(S \cdot action)$ is cycle-free, then execute $action$. Otherwise abort T and all transactions that depend on T, and remove the corresponding actions from S.
- **Timestamps:**
  Each transaction T is associated to a unique timestamp $Z(T)$.
  (thus, transactions can be seen as ordered).
  Let S the current (partial) schedule and $action$ the next action of some transaction T.
  If for all transactions T' that have executed an action a' that is in conflict with $action$, $Z(T') \leq Z(T)$, then execute $action$. Otherwise abort T (T "comes too late") and all transactions that depend on T, and remove the corresponding actions from S.

Scheduling Strategies (Cont'd)

- **Optimistic Strategies:**
  (weaker than Timestamp – Assumption: "there is no conflict")
  Let S the current (partial) schedule. A transaction T is **active** in S, if an action of T is contained in S, and T is not yet completed.
  Let $readset(T), writeset(T)$ the set of objects that have been read/written by a transaction T.
  Let $action$ the next action, and T the corresponding transaction.
  Execute $action$ and update $readset(T), writeset(T)$.
  If $action$ is the final action in T, then check the following:
  - if for any other active transaction T':
    * $readset(T) \cap writeset(T') \neq \emptyset$,
    * $writeset(T) \cap writeset(T') \neq \emptyset$,
    * $writeset(T) \cap readset(T') \neq \emptyset$.

  then abort T and all transactions that depend on T, and remove the corresponding actions from S.
- **2-Phase Lock:** Can we enforce the above criterion?

6.5 Locks

- access to database objects is administered by **locks**
- Transactions need/hold locks on database objects:
  if T has a lock on A, T is has a privilege to use this object
- privileges allow for read-only, or read/write access to an object:
  - Read-privilege: RLOCK ($L_R X$)
  - Read and write-privilege: WLOCK ($L_W X$)
- operations:
  - LOCK X (LX): apply for a privilege for using X.
  - UNLOCK X (UX): release the privilege for using X.
- lock- and unlock operations are handled like actions and belong to the action sequence of a transaction.
- each action of a transaction must be inside a corresponding pair of lock-unlock-actions.
  (i.e., no action without having the privilege)

Example 6.5

Consider again Example 6.2: $T = RA\ WA\ RB\ WB$

Possible handling of locking actions:

- $T = LA\ RA\ WA\ UA\ LB\ RB\ WB\ UB$
- $T = LA\ LB\ RA\ WA\ RB\ WB\ UA\ UB$
- $T = LA\ RA\ WA\ LB\ RB\ WB\ UA\ UB$
- $T = LA\ RA\ WA\ LBUA\ RB\ WB\ UB$

□

LOCKING POLICIES

Locking policies (helping the scheduler) must guarantee correct execution of parallel transactions.

- privileges are given according to a **compatibility matrix**:

  Y: requested privilege can be granted

  N: requested privilege cannot be granted

- if there is only one privilege ("use an object"):

  requested privilege \ granted privilege (for the same object):

	LOCK
LOCK	N

- if read and write privilege are distinguished:

	RLOCK	WLOCK
RLOCK	Y	N
WLOCK	N	N

  i.e., multiple transactions *reading* the same object are allowed.

PROBLEMS

- **Livelock**: It is possible that a transaction never obtains a requested lock (if always other transactions are preferred).
  Solution: e.g., first-come-first-served strategies

- **Deadlock**: during execution, deadlocks can occur:

  Transactions: T_1: LOCK A; LOCK B; RA WA RB WB UNLOCK A,B;

  T_2: LOCK B; LOCK A; RA WA RB WB UNLOCK A,B;

  Execution: T_1: LOCK A

  T_2: LOCK B

  Deadlock: no transaction can proceed.

Avoiding and resolving deadlocks

- each transaction applies for all required locks when starting (in an atomic action).
- a linear ordering of objects. Privileges must be requested according to this ordering.
- maintenance of a **waiting graph** between transactions: The waiting graph has an edge $T_i \to T_j$ if T_i applies for a privilege that is hold/blocked by T_j.
  - a deadlock occurs exactly if the waiting graph is cyclic
  - it an be resolved if one of the transactions in the cycle is aborted.

Note: locks alone do not yet guarantee serializability.

Example 6.6
Consider again Example 6.2 where T_1 and T_2 are extended with locks:

$T = LA\ RA\ WA\ UA\ LB\ RB\ WB\ UB$

Consider Schedule S_4 (which was not serializable):

$$S_{4L} = \quad L_1A\ \ R_1A\ \ W_1A\ \ U_1A\quad L_2A\ \ R_2A\ \ W_2A\ \ U_1A$$
$$L_2B\ \ R_2B\ \ W_2B\ \ U_2B\ \ L_1B\ \ R_1B\ \ W_1B\ \ U_1B$$ □

Only correct use and policies do.

We need a protocol/policy that – if satisfied – *guarantees* serializability.

2-PHASE LOCKING PROTOCOL (2PL)

"After the first UNLOCK, a transaction must not execute any LOCK."

i.e., each transaction has a **locking phase** and an **unlocking phase**.

Example 6.7
Consider again Example 6.5:

Which transactions satisfy 2PL?

- *T = LA RA WA UA LB RB WB UB (no)*
- *T = LA LB RA WA RB WB UA UB (yes)*
- *T = LA RA WA LB RB WB UA UB (yes)*
- *T = LA RA WA LB UA RB WB UB (yes!)* □

The last LOCK-operation of a transaction T defines T's **locking point**.

Theorem 6.3
The 2-Phase-Locking Protocol guarantees serializability. □

Proof: Consider a schedule S of a set $\{T_1, T_2, \ldots\}$ of two-phase transactions.

Assume that S is not serializable, i.e., $CG(S)$ contains a cycle, w.l.o.g. $T_1 \to T_2 \to \ldots \to T_k \to T_1$. Then, there are objects $A_1, \ldots, A_k$ such that

$$
\begin{aligned}
S = \ldots\ & (W_1A_1)\ \ U_1A_1\ \ \ldots\ L_2A_1\ \ (R_2A_1)\ \ \ldots \\
S = \ldots\ & (W_2A_2)\ \ U_2A_2\ \ \ldots\ L_3A_2(R_3A_2)\ \ \ldots \\
& \vdots \\
S = \ldots\ & (W_{k-1}A_{k-1})\ \ U_{k-1}A_{k-1}\ \ \ldots\ L_kA_{k-1}\ \ (R_kA_{k-1})\ \ \ldots \\
S = \ldots & (W_kA_k)\ \ U_kA_k \ldots\ L_1A_k\ \ (R_1A_k) \ldots
\end{aligned}
$$

Let l_i the locking point of T_i. Then, the above lines imply that l_1 is before l_2, that is before l_3 etc, and l_{k-1} before l_k, that is before l_1. Impossible.

Properties of 2PL

2-Phase locking is optimal in the following sense:

For every non 2-phase transaction T_1 there is a 2-phase transaction T_2 such that for T_1, T_2 there exists a non-serializable schedule.

Example 6.8
Consider the non-2PL transaction from Example 6.7 and a 2PL transaction

$$T_1 = L_1A \ R_1A \ W_1A \ U_1A \ L_1B \ R_1B \ W_1B \ U_1B$$
$$T_2 = L_2A \ L_2B \ R_2A \ W_2A \ R_2B \ W_2B \ U_2A \ U_2B$$

The following schedule S (= S_4 from Examples 6.2) which has been shown not to be serializable:

$$S = \ L_1A \ R_1A \ W_1A \ U_1A \ L_2A \ L_2B \ R_2A \ W_2A$$
$$R_2B \ W_2B \ U_2A \ U_2B \ L_1B \ R_1B \ W_1B \ U_1B$$

□

Properties of 2PL (Cont'd)

"optimal" does *not* mean that every serializable schedule can also occur under 2-phase locking:

Example 6.9
The schedule S

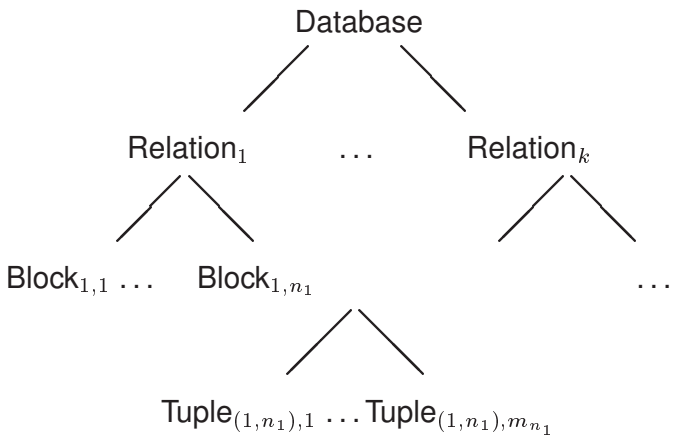
$$S = R_1A \ R_2A \ W_2A \ R_3B \ W_3B \ W_1B$$

is serializable, but there is no way to add LOCK/UNLOCK actions to T_1, T_2 that satisfy the 2PL requirement such that S is a schedule of these transactions.

□

LOCKING STRUCTURED DATA

- database consists of relations that are stored in blocks that contain tuples.



- find a compromise between maximal parallelism and number of locks.
- transactions that use all tuples of a relation: lock the relation
- transactions that lock only some tuples of a relation: lock the tuples.

LOCKING GRANULARITY

Having only tuple-locks and 2PL can still lead to non-serializable schedules:

Example 6.10
Consider again Slide 165.

T_1 *computes the sum of the population of all provinces of Germany – reading all tuples. Thus, at the beginning it locks all (existing) tuples.* T_2 *adds a new province and adapts the overall population.*

The schedule given on Slide 165 is still possible. □

Solution: Locking of complete tables, key areas, depending on predicates, or indexes.

Consequence: if the set of database objects changes dynamically, a conflict-based serializability test is not sufficient.

LOCKING IN THE SQL2-STANDARD

Serializability is enforced as follows:

- every transaction does only see updates by committed transactions
- no value that has been read/written by T can be changed by any other transaction before committing/aborting T.
  That means, “locks” are released only *after* commit (**strict 2-Phase Locking**).
- if T has read a set of tuples defined by some search criterion, this *set* cannot be changed until T is committed or aborted. (this excludes the phantom-problem)

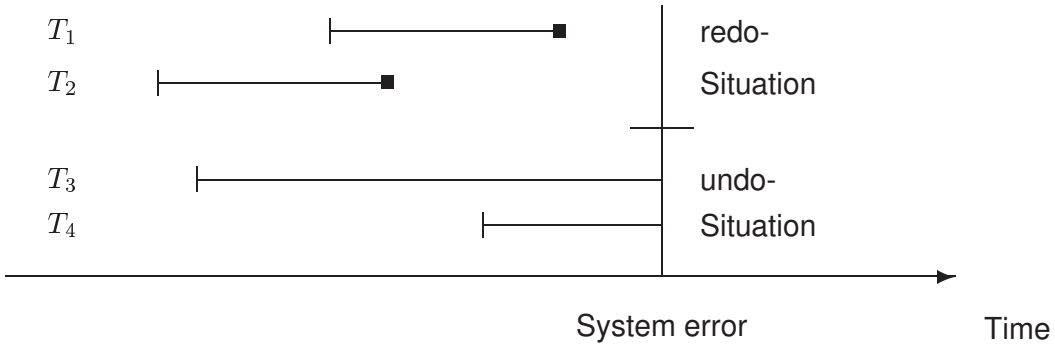
6.6 Safety: Error Recovery

What (dangerous) errors can happen to a database system?

- **Transaction errors**
  local, application-semantical error situations
  - error situation in the application program
  - user-initiated abort of transaction
  - violation of system restrictions (authentication etc)
  - resolving of a deadlock by aborting a transaction.
- **System errors**
  runtime environment crashes completely
  - hardware errors (main memory, processor)
  - faulty values in system tables that cause a software crash
- **Media crashes**
  database backend crashes
  crash of secondary memory (disk head errors ...)

Assumption: Transactions satisfy **strict 2PL**.

REDO- AND UNDO-SITUATIONS



- **redo-Situation:**
  A transaction has committed, and an error occurs.
- **undo-Situation:**
  A transaction already writes to the database before committing. During execution, an error occurs.

  undo-avoiding:

  write actions are only written to the database ("materialized") *after* committing. Before, they are cached in a special buffer.

DATABASE LOG

The database system maintains a **log** (journal) where all changes in the database and all state changes of transactions (BOT/EOT) are recorded.

Entries (sequential):

(1) at begin of transaction: $(T, begin)$

(2) if a transaction T executes WX:

$(T , X , \quad X_{old} , \quad X_{new})$

value of X written by T **(after image)**

value of X before WX **(before image)**

(3) at commit: $(T, commit)$

(4) at abort: $(T, abort)$

Example 6.11
(write-immediate, no undo-avoiding)

T_1:	*LA RA*		*WA CO UA*			
T_2:		*LB RB*		*LA RA WB*	*CO UA UB*	
T_3:				*LC RC*	*WC*	
Database:						
A:	A_0	$f_1(A_0)$				
B:	B_0		$f_2(f_1(A_0), B_0)$			
C:	C_0				$f_3(C_0)$	
Buffers:						
T_1:		$A : f_1(A_0)$				
T_2:			$B : f_2(f_1(A_0), B_0)$			
T_3:					$C : f_3(C_0)$	

Log:
$(T_1, A, A_0, f_1(A_0)), (T_1, CO), (T_2, B, B_0, f_2(f_1(A_0), B_0)), (T_2, CO), (T_3, C, C_0, f_3(C_0))$ □

Transaction Errors

Consider a transaction T that is aborted before reaching its COMMIT phase.

If undo-avoiding is used, no error handling is required (simply discard its log entries),

Otherwise, process log file backwards up to $(T, begin)$ and materialize for every entry (T, X, X_{old}, X_{new}) the (before-)value X_{old} for X in the database.

Note that due to strict 2PL, no other transaction could read values that have been written by T!

System Errors

Restart-Algorithm (without savepoints, for strict 2PL)

- $redone := \emptyset$ and $undone := \emptyset$.
- process the logfile backwards until end, or $redone \cup undone$ contains all database objects.

  For every entry (T, X, X_{old}, X_{new}):

  If $X \notin redone \cup undone$:
  - If the logfile contains $(T, commit)$ (then redo), then write X_{new} into the database and set $redone := redone \cup \{X\}$.
  - Otherwise (undo) write X_{old} into the database and set $undone := undone \cup \{X\}$. ("undo once" only correct for **strict 2PL**!)

  If undo-avoiding is used, no undo is required.

Example 6.12

Consider again Example 6.11.

(write-immediate, no undo-avoiding)					*Sys.error*	*state after restart*
T_1:	*LA RA*		*WA CO UA*			
T_2:		*LB RB*		*LA RA WB*	*CO UA UB*	
T_3:				*LC RC*	*WC*	
Database:						
A:	A_0		$f_1(A_0)$			$f_1(A_0)$
B:	B_0			$f_2(f_1(A_0), B_0)$		$f_2(f_1(A_0), B_0)$
C:	C_0				$f_3(C_0)$	C_0
Buffers:						
T_1:			$A : f_1(A_0)$			
T_2:				$B : f_2(f_1(A_0), B_0)$		
T_3:					$C : f_3(C_0)$	

Log:

$(T_1, A, A_0, f_1(A_0)), (T_1, CO), (T_2, B, B_0, f_2(f_1(A_0), B_0)), (T_2, CO), (T_3, C, C_0, f_3(C_0))$ □

Logging Requirements

Log granularity:

the log-granularity must be finer than the lock granularity. Otherwise, redo or undo can also delete effects of other transactions than intended.

Write-ahead:

before a write action is materialized in the database, it must be materialized in the log file (materialized means that it must actually be written to the DB, not only to a buffer – which could be lost)

Savepoints

... processing the log backwards ...

until the most recent **savepoint**.

Generation of a Savepoint

- Do not begin any transaction, and wait for all transactions to finish (COMMIT or ABORT).
- Materialize all changes in the database (force write caches).
- write *(checkpoint)* to the logfile

Media Crash

Solution: Redundancy

Strategy 1: keep a complete copy of the database (incl log)

Probability that both are destroyed at the same time is low (keep them in different computers in different buildings ...)

Writing of a tuple to the database means to write it also in the copy. Copy is written only after write to original is confirmed to be successful (otherwise e.g. an electrical breakdown kills both).

Strategy 2: periodical generation of an archive database (dump).

After generation of the dump, $(archive)$ is written to the logfile.

In case of a media crash, restart as follows:

- Load the dump.
- apply restart-algorithm only wrt. redo of completed (committed) transactions back to the $(archive)$ entry.