# 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 (complete! – cf. atomicity) 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, *transactions* are *logically atomic*, but several transactions may be executed in an *interleaved* manner (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.* □

### Atomicity

A transaction is *logically atomic* – even when executed interleaving with others

- all-or-nothing,
- potential rollback at the end,

$\Rightarrow$ requires isolation – other transactions must not use uncommitted written values (or also rolled back)

$\Rightarrow$ rollback based on logging (see Slide 309 ff.).

### Consistency

- Concept: check conditions only at the end of a transaction (COMMIT)
- Default in DB systems: Check after each atomic action
- Optional: declare CONSTRAINTs with DEFERRABLE INITIALLY DEFERRED to postpone checks.



## 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

# NESTED TRANSACTIONS [OPTIONAL]

- internal (hierarchical) structuring of a transaction into **subtransactions**
- subtransactions can be executed serially, synchronous parallel, or asynchronous parallel
- transaction satisfies ACID, subtransactions only A&I.

## Properties of Subtransactions

- atomicity
- consistency: not required – only for the root transaction
- isolation: required for rollback
- durability: not possible, since rollback of a superordinate (sub)transaction required also to rollback "committed" subtransactions



## Properties of Subtransactions (Cont'd)

- Commit: the local commit of a subtransaction makes its effects accessible only for its superordinate transaction.
- root transaction commits if all immediate subtransactions commit.
- rollback: if some (sub)transaction is rolled back, all its subtransactions are rolled back recursively (even when they committed locally)
- visibility: all updates of a subtransaction become visible to its superordinate transaction when it commits.
  All objects that are kept by a transaction are accessible for its subtransactions.
  Effects are not visible for sibling transactions.
- above: "closed nested transactions"
- weaker visibility/isolation requirements: "open nested transactions"
  require more complex rollback mechanisms

## 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 are 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

For multiuser aspects, consider a scenario where a high number of short and long transactions has to be processed:

(Online) Banking

A bank maintains branches at several cities; at each city multiple customers have accounts. Customers are doing money transfers, cash withdrawals at ATMs (german: Geldautomaten); and the bank computes the yearly interest rate (german: Zinsen) always on January 1st.

Consider the following relations:

- Account: Name, City, Amount
- Branch: City, Total
  where "Total" contains the sum of all accounts at that place.

## Lost update

| money transfer $A \to B$ | $B$ taking cash at the ATM |
|---|---|
| SELECT amount INTO a<br>FROM Accounts WHERE name = 'Alice' | |
| a := a - 100; | |
| UPDATE Accounts<br>SET amount = a WHERE name = 'Alice' | |
| SELECT amount INTO b<br>FROM Accounts WHERE name = 'Bob' | |
| | SELECT amount INTO c<br>FROM Accounts WHERE name = 'Bob' |
| b := b + 100; | |
| | c := c - 200; |
| UPDATE Accounts<br>SET amount = b WHERE name = 'Bob' | |
| | UPDATE Accounts<br>SET amount = c WHERE name = 'Bob' |

- the money transfer (first update) is lost



## Dirty Read

| money transfer $A \to D$ fails | $A$ taking cash at the ATM |
|---|---|
| SELECT amount INTO a<br>FROM Accounts WHERE name = 'Alice' | |
| a := a - 100; | |
| UPDATE Accounts<br>SET amount = a WHERE name = 'Alice' | |
| SELECT amount INTO d<br>FROM Accounts WHERE name = 'Dave' | |
| · · · search and wait · · · | |
| | SELECT amount INTO c<br>FROM Accounts WHERE name = 'Alice' |
| | c := c - 200; |
| | UPDATE Accounts<br>SET amount = c WHERE name = 'Alice' |
| Dave does not have an account here! | |
| ABORT (i.e., ROLLBACK) | |
| (what must be done now?) | |

- The second transaction reads and uses a value that is later undone

## Non-repeatable Read

| Sum of Accounts | $A$ and $C$ taking cash at ATM |
|---|---|
| sum := 0 | |
| SELECT amount INTO x<br>FROM Account WHERE name = 'Alice' | |
| sum := sum + x | |
| | UPDATE Accounts |
| | SET amount = amount - 100<br>WHERE name = 'Alice' |
| SELECT amount INTO x<br>FROM Account WHERE name = 'Bob' | |
| sum := sum + x | |
| | UPDATE Accounts |
| | SET amount = amount - 200<br>WHERE name = 'Carol' |
| SELECT amount INTO x<br>FROM Account WHERE name = 'Carol' | |
| sum := sum + x | |

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



## Phantom

- sum of accounts equals total for each branch?

| Check sum of population by branch | Insert new account |
|---|---|
| SELECT SUM(amount) INTO sum<br>FROM Account WHERE city = 'Frankfurt' | |
| | INSERT INTO Accounts (name, city, amount)<br>VALUES ('Dave', 'Frankfurt', 1000) |
| | UPDATE Branches<br>SET total = total + 1000<br>WHERE city = 'Frankfurt' |
| SELECT total INTO x<br>FROM Branches WHERE city = 'Frankfurt' | |
| IF x $\neq$ sum THEN<br><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:
  A transaction must 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 | |
| $S_1$: serial | | $S_2$: not serializable | | $S_3$: serializable | | $S_4$: 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.

**Example 6.3**

*Consider again Example 6.2 for A=B=10;*
$T_1'$*: RA; A:=A*1.05; WA; RB; B:=B*1.05; WB (Yearly Interest Rate) and*
$T_2$*: RA; A:=A-10; WA; RB; B:=B+10; WB (Money Transfer).*
*Consider* $S_1$, $S_5 := T_2T_1$, $S_3$, *and* $S_4$.

□

## 6.4.1 Formalization of the Semantics of Transactions

- How to show that *for all* possible circumstances, a schedule is serializable?
- Theory & algorithms depend only on the READ and WRITE actions, not on the semantics of the computations in-between.

  (this would require theorem-proving instead of symbolic algorithms)

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



## ASIDE: BASIC NOTIONS OF FIRST-ORDER PREDICATE LOGIC

(you probably have learnt this in “Discrete Mathematics” or in “Formal Systems”)

- An first-order signature $\Sigma$ contains **function symbols** and **predicate symbols**, each of them with a given arity (function symbols with arity 0 are constants).
- The set of **ground terms** over $\Sigma$ is built inductively over the function symbols: for $f \in \Sigma$ with arity $n$ and terms $t_1, \ldots, t_n$, also $f(t_1, \ldots, t_n)$ is a term.
- A **first-order structure** $\mathcal{S} = (\mathcal{D}, I)$ over a signature $\Sigma$ consists of a nonempty set $\mathcal{D}$ (**domain**) and an interpretation $I$ of the signature symbols over $\mathcal{D}$ which maps
  - every constant $c$ to an element $I(c) \in \mathcal{D}$,
  - every $n$-ary function symbol $f$ to an $n$-ary function $I(f) : \mathcal{D}^n \to \mathcal{D}$,
  - every $n$-ary predicate symbol $p$ to an $n$-ary relation $I(p) \subseteq \mathcal{D}^n$.

## Logic Formalization of the Semantics of Transactions

- Let $\mathcal{D}$ denote the domain of the database objects.
- Consider a transaction $T$, with a write action $WX$ where $RY_1, \ldots, RY_k$ $k \geq 0$ are the read actions that are executed by $T$ before $WX$.
- The value written to $X$ by $WX$ is denoted by

$$f_{T,X}(Y_1, \ldots, Y_k)$$

where

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

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

- the functions $f_{T,X}$ abstract the calculation of the value of $X$ that is then written by $WX$ in $T$,
- their actual interpretation is given by the computation of the transaction.



## Application to Transactions and Schedules

- for every transaction and every schedule, the final values (call them $a_\infty, b_\infty, ...$) can be expressed in terms of the $t_{T,X}$ of the contributing transactions,
- the constants $a_0$, $b_0$ are interpreted by initial values,
- the actual interpretation of the functions is given by the transaction.

Consider the single transaction runs:

| | | |
|---|---|---|
| $T_1$: | $RA$ | read $a_0$ |
| | $A := A - 10$ | |
| | $WA$ | write $f_{T_1,A}(a_0)$ |
| | $RB$ | read $b_0$ |
| | $B := B + 10$ | |
| | $WB$ | write $f_{T_1,B}(a_0, b_0)$ |
| | | $a_\infty = f_{T_1,A}(a_0)$, $b_\infty = f_{T_1,B}(a_0, b_0)$ |

| | | |
|---|---|---|
| $T_1'$: | $RA$ | read $a_0$ |
| | $A := A * 1.05$ | |
| | $WA$ | write $f_{T_1',A}(a_0)$ |
| | $RB$ | read $b_0$ |
| | $B := B * 1.05$ | |
| | $WB$ | write $f_{T_1',B}(a_0, b_0)$ |
| | | $a_\infty = f_{T_1',A}(a_0)$, $b_\infty = f_{T_1',B}(a_0, b_0)$ |

both induce the same final term structure. The interpretations differ:
$T_1$: $f_{T_1,A}(A) = A - 10$, $f_{T_1,B}(A, B) = B + 10$,
$T_1'$: $f_{T_1',A}(A) = A * 1.05$, $f_{T_1',B}(A, B) = B * 1.05$.

## Application to Single Transactions

- for given transactions (i.e. a given interpretation of $f_{T,X}$), properties of the final values can formally be proven, e.g.,
  $T_1 : a_\infty + b_\infty = a_0 + b_0$,
  $T_1' : a_\infty + b_\infty = (a_0 + b_0) * 1.05$
- later/Exercise: intra-transactional optimization by interchanging non-conflicting operations of a transaction.



## Application to Schedules

**Example 6.4**

*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* $T_1T_2$ *(serial)* | | | *Schedule* $T_2T_1$ *(serial)* | | |
|---|---|---|---|---|---|
| $T_1:$ | $RA$ | $a_0$ | $T_2:$ | $RA$ | $a_0$ |
| | $WA$ | $f_{T_1,A}(a_0)$ | | $WA$ | $f_{T_2,A}(a_0)$ |
| | $RB$ | $b_0$ | | $RB$ | $b_0$ |
| | $WB$ | $f_{T_1,B}(a_0, b_0)$ | | $WB$ | $f_{T_2,B}(a_0, b_0)$ |
| $T_2:$ | $RA$ | $f_{T_1,A}(a_0)$ | $T_1:$ | $RA$ | $f_{T_2,A}(a_0)$ |
| | $WA$ | $f_{T_2,A}(f_{T_1,A}(a_0))$ | | $WA$ | $f_{T_1,A}(f_{T_2,A}(a_0))$ |
| | $RB$ | $f_{T_1,B}(a_0, b_0)$ | | $RB$ | $f_{T_2,B}(a_0, b_0)$ |
| | $WB$ | $f_{T_2,B}(f_{T_1,A}(a_0), f_{T_1,B}(a_0, b_0))$ | | $WB$ | $f_{T_1,B}(f_{T_2,A}(a_0), f_{T_2,B}(a_0, b_0))$ |

□

- for a given interpretation, the evaluation of the terms yields the final values,
- the terms themselves additionally encode the data flow through the schedule.

## EQUIVALENCE OF SCHEDULES

**Definition 6.2**
*Two schedules $S, S'$ (of the same set of transactions) are* equivalent*, if for every initial state, corresponding atomic actions read/write the same values in $S$ and $S'$.* □

**Corollary 6.1**
*For two equivalent schedules $S$ and $S'$ executed on the same initial state, $S$ and $S'$ generate the same final states.* □

Proof: consider the last write actions for each data item.

- according to the Definition 6.2, equivalence can only be checked by investigating both schedules step-by-step.
- In the above formalization, this is encoded into the (final) terms: the execution of a schedule is traced symbolically
  ("Herbrand interpretation" – every term is interpreted "as itself").



## EQUIVALENCE OF SCHEDULES

**Exercise 6.1**
*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.* □

**Example 6.5 (Solution of Exercise 6.1)**

*Consider again the transactions* $T_1 = RA\ WA\ RB\ WB$ *and* $T_2 = RA\ WA\ RB\ WB$ *and the schedules* $S_2$ *and* $S_4$*. Let the initial state again be given by values* $a_0, b_0$*.*

| *Schedule* $S_2$ | | *Schedule* $S_4$ | |
|---|---|---|---|
| $RA$ | $a_0$ | $RA$ | $a_0$ |
| $RA$ | $a_0$ | $WA$ | $f_{T_1,A}(a_0)$ |
| $WA$ | $f_{T_1,A}(a_0)$ | $RA$ | $f_{T_1,A}(a_0)$ |
| $WA$ | $f_{T_2,A}(a_0)$ | $WA$ | $f_{T_2,A}(f_{T_1,A}(a_0))$ |
| $RB$ | $b_0$ | $RB$ | $b_0$ |
| $RB$ | $b_0$ | $WB$ | $f_{T_2,B}(f_{T_1,A}(a_0), b_0)$ |
| $WB$ | $f_{T_1,B}(a_0, b_0)$ | $RB$ | $f_{T_2,B}(f_{T_1,A}(a_0), b_0)$ |
| $WB$ | $f_{T_2,B}(a_0, b_0)$ | $WB$ | $f_{T_1,B}(a_0, f_{T_2,B}(f_{T_1,A}(a_0), b_0))$ |

*For* $S_3$*, the terms are the same for every R/W as for* $S_1$*.*
*For* $S_4$*, the blue-red-blue "shows" that there can be no serial schedule that generates the same terms.* □



## A Step Towards More Abstraction

Summary and conclusions:

- the $f$s are abstractions for the actual functions/computations of the transactions,
- we are actually not interested at all, what the $f$s are,
- but (mainly) in the term structure and the data flow (indicated by the colors above),
- the "history" of a data item is described by the $f$-terms.

$\Rightarrow$ find another way to represent how information flows and "who reads and writes what values".

### 6.4.2 Theoretical Investigations

Consider a schedule $S$ together with two additional distinguished transactions $T_0, T_\infty$:
$T_0$ generates the initial state, and $T_\infty$ 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.
- $T_\infty$ 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 6.2**
*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 $T_i$)

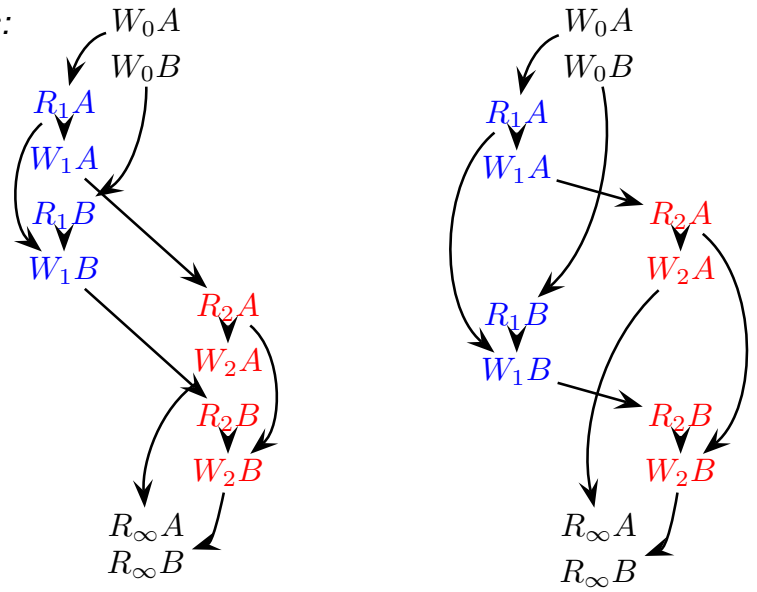
A transaction $T'$ is **dependent** of a transaction $T$, if in $S$, either $T'$ reads a value that has been written by $T$, or by a transaction that is dependent on $T$.

**Example 6.6**

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

*Consider the serial schedule* $T_1T_2$ *and* $S_3 = R_1A\ W_1A\ R_2A\ W_2A\ R_1B\ W_1B\ R_2B\ W_2B$.

*Dependency graphs:*




□

**Theorem 6.1**

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

□



... and now to the

Proof [Optional]

Each transaction $T$ with $n$, $n \geq 1$ write actions on $A_1, \ldots, A_n$ induces a set $F_T = \{f_{T,A_1}, \ldots, f_{T,A_n}\}$ of function symbols that are used for representing the computations associated with the write actions.

Given a domain $\mathcal{D}$, every transaction also induces an interpretation

$$\mathcal{S}_\mathcal{T} = (\mathcal{D}, I_{F_T}) \text{ such that each } I(f_{T,A_i}) \text{ is a mapping } \mathcal{D}^{k_i} \to \mathcal{D}\ .$$

Analogously, the interpretation of a set $T_1, \ldots, T_m$ of transactions has the form $\mathcal{S} = (\mathcal{D}, F_{T_1} \cup \ldots \cup F_{T_m})$.

Assume an action $a$ of a schedule $S$. If $a$ is a write action, then $a_S(I)$ is the value that is written by $a$ in $S$ under the interpretation $I$. If $a$ is a read action, then $a_S(I)$ is the value that is read by $a$.

For a node $a$ of the D-graph $DG(S)$, the **restriction** of $DG(S)$ to $a$ and its predecessors is denoted by $pred_S(a)$ – (this is the portion of the graph consisting of all actions that contribute to the value that is read/written by $a$).

## Proof (Cont'd)

"$\Leftarrow$": We show that for all actions $a$ in $S$,

$$pred_S(a) = pred_{S'}(a) \Rightarrow a_S(I) = a_{S'}(I)$$

for arbitrary interpretations $I$ by induction over the number of nodes in $pred_S(a)$.

Assume that $a$ is an action in a transaction $T$ to a database object $x$.

- $pred_S(a)$ contains a single node. Then, $a$ is this node.
  - $a$ cannot be a read action, as any read action $RA$ would have at least a write action $W_0$ in $T_0$ as predecessor.
  - if $a$ is a write action on $A$, $f_{T,A}$ is a constant function (depending on no input/original values) and thus, $a_S(I) = a_{S'}(I)$ for all $I$.
- $pred_S(a)$ contains more than a single node. Because of $pred_S(a) = pred_{S'}(a)$, $a$ has the same predecessors $b_1, \ldots, b_k$ in both graphs. By induction hypothesis, for each of them, $b_S(I) = b_{S'}(I)$.
  - if $a$ is a read action, the conclusion $a_S(I) = a_{S'}(I)$ is again trivial.
  - if $a$ is a write action on $A$,
    $a_S(I) = f_{T,A}(b_{1S}(I), \ldots, b_{kS}(I)) = f_{T,A}(b_{1S'}(I), \ldots, b_{kS'}(I)) = a_{S'}(I)$.

Thus, in both sequences, the same values are read and written.



## Proof (Cont'd)

"$\Rightarrow$":

Since $S$ and $S'$ are assumed to be equivalent, all transactions in $S$ and $S'$ read the same values for arbitrary interpretations $I$.

Consider the (Herbrand-) [that means, using uninterpreted ground terms] interpretation $H$ to the transactions in $S$:

- $\mathcal{D} = \{f_{T_0,A_1}, f_{T_0,A_2}, \ldots, f_{T_1,A_1}(\ldots, f_{T_0,A_1}, \ldots), \ldots,$
  $f_{T_2,A_1}(\ldots, f_{T_0,A_1}, \ldots, f_{T_1,A_1}(\ldots, f_{T_0,A_1}, \ldots), \ldots), \ldots\}$
  is the set of (ground) terms built over the symbols that are assigned to the write actions.
- $f_{T,A} : \mathcal{D}^k \to \mathcal{D}$: applying $f_{T,A}$ to values $v_1, \ldots, v_k$ yields the term $f_{T,A}(v_1, \ldots, v_k)$.

For every action $a$ in $S$ and $S'$, $a_S(H)$ and $a_{S'}(H)$ encode $pred_S(v)$ and $pred_{S'}(v)$, resp.

(e.g., if for a write action $a = W_1B$, $a_S(H) = f_{T_1,B}(a_0, f_{T_2,B}(f_{T_1,A}(a_0), b_0))$, then it has a predecessor $R_1A$ that read $a_0$, and a predecessor $R_1B$ that read the value $f_{T_2,B}(f_{T_1,A}(a_0), b_0)$ written by $W_2B$ that in course had (i) a predecessor $R_2A$ that read value $f_{T_1,A}(a_0)$ written by $W_1A$ that in turn had a predecessor $R_1A$ that read $a_0$, and (ii) a predecessor $R_2B$ that read $b_0$).

Thus, since $v_S(H) = v_{S'}(H)$, $DG(S) = DG(S')$.

## NEXT STEP TOWARDS MORE ABSTRACTION

- we are even not interested in the Dependency Graph, only in the question whether there is a serial schedule with the same DG.

**Example 6.7**
*Consider the DG of $S_4$ Example 6.2 (Example/Exercise)* □

- intra-transaction edges are not relevant
  (for a given transaction they are the same in all DGs),
- edges *between* transactions are important
  (see above example),
- some other relationships between transactions are also important.

$\Rightarrow$ they tell, what conditions an equivalent serial schedule must satisfy!

- ... if they are satisfiable, there is an equivalent serial schedule (or several of them).



## THEORY: EQUIVALENCE CLASSES OF SCHEDULES

### Recall from Discrete Mathematics

A binary relation $\sim$ is an equivalence relation on a set $X$ if it is

- reflexive: $x \sim x$ for every $x \in X$,
- symmetric: $x \sim y \Rightarrow y \sim x$ for every $x, y \in X$
- transitive $x \sim y \wedge y \sim z \Rightarrow x \sim z$ for every $x, y, z \in X$.

### Equivalence Classes

For an equivalence relation $\sim \subseteq X \times X$, the equivalence class $[x]$ is defined as

$$[x] := \{y \in X | x \sim y\}$$

Note: two equivalence classes are either the same, or disjoint.

## Equivalence Classes of Schedules and Serializable Schedules

On the set of schedules, let $\sim$ be defined as $S \sim S'$ if $DG(S) = DG(S')$. A schedule $S$ is serializable, if $S \in [S']$ for a serial schedule $S'$.

The following properties hold:

- given $n$ transactions, there are at most $n!$ equivalence classes of serializable schedules,
- for two serial schedules, $[S_1] = [S_2]$ is possible (when two or more transactions have no conflicts at all),
- there are many more equivalence classes of non-serializable schedules.



## Neighboring Schedules

**Definition 6.3**
*Two schedules $S, S'$ (of the same set of transactions) are* neighbors *if $S'$ can be obtained from $S$ by exchanging a single pair of atomic actions $a_1, a_2$.* □

Note:

- for a given set of transactions $T$, $a_1, a_2$ above must belong to different transactions to obtain a valid schedule of $T$.
- Aside: if exchanging actions of *the same* transaction, the approach is applicable to intra-transaction optimization:

  Actions in a transaction can be exchanged if the D-Graph is not effected (e.g., $R_iA$ and $R_iB$).

## When are Neighboring Schedules Equivalent?

Let $S = S_1 a_i a_j S_2$ and $S' = S_1 a_j a_i S_2$ be neighboring schedules.

Consider each pair of types of actions possible for $(a_i, a_j)$.

- obviously, actions on *different* data items can be exchanged without effecting the D-Graph.

**RR:** $R_i A, R_j A$: no change.

**WR:** $W_i A, R_j A$: WR is an edge in the D-Graph, exchanging the actions removes this edge and adds an edge from the preceding $W_j A$ to $R_j A$.

**RW:** $R_i A, W_j A$: symmetric. RW represents a "no-edge" in the DG.

**WW:** $W_i A, W_j A$: For the next $R_k A$ (if no $W_\ell$ is in-between [this condition will become relevant later – note also that $T_\infty$ is needed here]), there is an edge in the D-Graph $W_j A \to R_k A$; after exchanging, there is an edge $W_i A \to R_k A$.

In the RW/WR/WW cases the D-Graph is different from before, $S \not\sim S'$.

The respective pairs of actions $a_i, a_j$ determine a constraint (that distinguishes $S$ from $S'$ and $[S]$ from $[S']$) on the equivalent serial schedule that $T_i$ must be executed before $T_j$.



## Conflict Graph: Idea

Every schedule can be characterized wrt. the equivalence class it belongs to by these "borders" between their member sets.

- the constraints state a topological order on the set $T$ of transactions,
- if the graph is cyclic, then the set of constraints is not satisfiable (= there is no equivalent serial schedule).

  Note that $[S]$ then also exists, but does (usually; cf. later) not contain any serial schedule; only under certain conditions, there may be an equivalent serial schedule.
- if the graph does not contain a cycle, the constraints can be interpreted as a topological order that characterizes the equivalence class.

## CONFLICT GRAPH: DEFINITION

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 given 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**
*If the conflict graph $CG(S)$ of a schedule $S$ is cycle-free, then $S$ is serializable.* □



### Conflict Graph Theorem: Proof

Assumed: Since $CG(S)$ is cycle-free.

Interpret $CG(S)$ as a topological order of the nodes (i.e., of the transactions).
Short: let $S'$ a serial schedule according to this ordering. Then, $DG(S) = DG(S')$ and $[S] = [S']$.

Long:

Let $CG^*(S)$ denote the transitive closure of $CG(S)$.

For any serial $S' = T_{i_1} \ldots T_{i_n}$ over $T_1, \ldots, T_n$ (i.e., $\{i_1, \ldots, i_n\} = \{1, \ldots, n\}$), let

$\leq_{S'} := \{(n, m) | T_n$ occurs before $T_m$ in $S'\}$

$S' \sim S \Leftrightarrow CG^*(S) \subseteq \leq_{S'}$ (i.e., if the orderings are consistent).

As $CG(S)$ is noncyclic, it is a (satisfiable) topological ordering and such an $S'$ exists.

### Remarks

Note that the ordering given by $CG(S)$ may be incomplete, i.e. there can be serial $S' \neq S''$ both in the same equivalence class: $[S] = [S'] = [S'']$.

## CONFLICT GRAPHS

**Example 6.8**
*Consider the CGs of $S_1$, $S_3$, $S_4$ from Example 6.2.* □

**Example 6.9**
*Consider the schedule*

$$S = R_1A\ W_1A\ R_2A\ R_3A\ R_2B\ R_3C\ W_3C\ W_2B\ .$$

*Draw the Conflict Graph, interpret it as a topological order and give all equivalent serial schedules. Draw the DG of $S$ and the DGs of the equivalent serial schedules.* □



## NEIGHBORING SCHEDULES

- Given a serial schedule, equivalent schedules can be constructed by considering neighboring schedules:
  "It is allowed to postpone action $a_i$ of $T_i$ and instead already process action $a_j$ from $T_j$?"
  (cf. Schedule $S_3$ in Example 6.2)
- define $\sim_1$ (one-change-equivalence) by

$$S_1 \sim_1, S_2 \ :\Leftrightarrow\ S_1 \text{ and } S_2 \text{ are neighbors and } S_1 \sim S_2$$

- define $\sim_n$ ($n$-change-equivalence) by

$$S_1 \sim_n, S_2 \ :\Leftrightarrow\ \text{there are } S_1, S_2, \ldots, S_n \text{ such that } S_i \sim_1 S_{i+1} \text{ for all } 1 \leq i < n$$

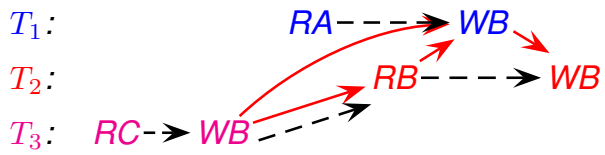
- Obviously, $\sim \subseteq \bigcup_{n \in \mathbb{N}} \sim_n$.
  (When) does equality hold?

## Blind Writes

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

**Example 6.10**

*Consider the following schedule $S$:*




*red lines: conflicts inducing the CG*

*black, dashed lines: dependency graph*

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

*Nevertheless, $S' = T_1T_3T_2$ is an equivalent (i.e., with the same dependency graph) serial schedule:*

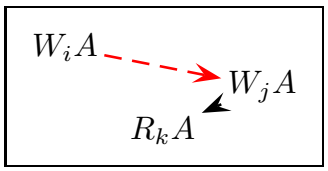

*with conflict graph (1,3), (3,2).*

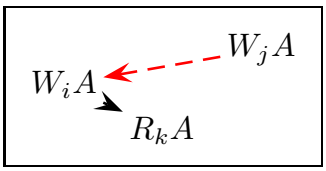
*Why this? $W_1B$ is not used anywhere in $S$. It is also not used in $S'$ (since $T_3$ does not read it before writing $B$). $W_3B$ and $W_1B$ are "Blind Writes" (a transaction does a $WX$ without a $RX$ before).* □



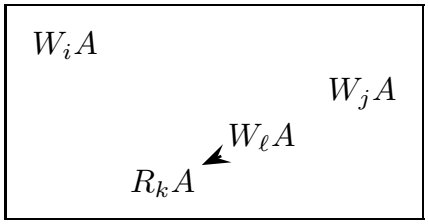
## Closer Look at WW Conflicts

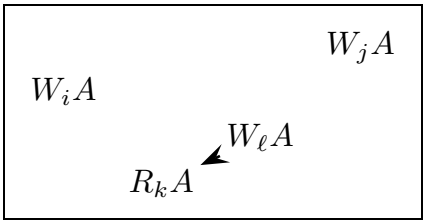
Consider again Slide 275 and $W_iA$ $W_jA$-conflicts:




If there is another $W_\ell A$ before the next $R_kA$, the D-Graph is *not* changed when interchanging $a_i$ and $a_j$:

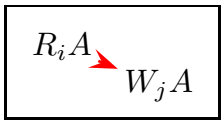


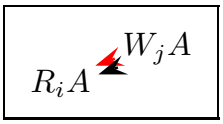




- A write $W_jA$ "cuts" the data flow from the preceding $W_iA$.
- WW-conflicts where the second write is never read can be ignored.
- the above fragments can only be completed to equivalent serializable schedules if $W_\ell A$ and either $W_iA$ or $W_jA$ are blind writes.

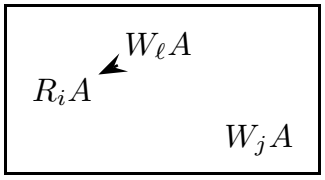
## CLOSER LOOK AT RW CONFLICTS

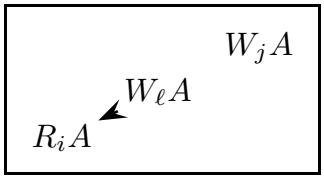
Consider again Slide 275 and $R_iA\ W_jA$-conflicts:




Exchanging $R_iA$ and $W_jA$ leads to a dataflow. If $W_jA$ is not put immediately before, but much earlier, the original dataflow is (locally) unchanged:




- the above fragments can only be completed to equivalent serializable schedules if $W_jA$ is followed by a blind write in both cases (which is exactly the case as in Example 6.10).



## EQUIVALENT SCHEDULES IN PRESENCE OF BLIND WRITES

- Example 6.10 shows that in presence of two blind writes a completely different schedule can be equivalent.
- WR-conflicts represent actual data flow – they must be the same in the corresponding serial schedule.
- WW-conflicts and RW-conflicts can be ignored under certain conditions (needing at least two blind writes on the corresponding data item)
- In the RW case, there is no path wrt. the "neighborship" relation from $S'$ to $S$ that stays inside $[S] = [S']$, i.e., $[S] \not\sim_n [S']$ for any $n$.

C-Graph-Serializability

**Definition 6.4**
*A schedule is **C-serializable** (conflict-serializable) if its C-Graph is cycle-free.* □

**Theorem 6.3**
*If for a set $\mathcal{T}$, there are no "blind writes", i.e., for each $T \in \mathcal{T}$,*

$$T = \ldots WA \ldots \Longrightarrow T = \ldots RA \ldots WA \ldots,$$

*then every schedule $S$ over $\mathcal{T}$ is serializable if and only if $S$ is C-serializable.* □

Proof: Exercise.

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



## 6.4.3 More Detailed Serializability Theory [Optional]

The C-Graph is more restrictive than necessary (cf. the above example):

For a more liberal criterion, consider only the following situation:

$$S = \ldots W_i A \ldots R_j A \ldots$$

and there is no write action on $A$ between $W_i$ and $R_j$.

- In every equivalent serial schedule, $T_i$ precedes $T_j$,
- if $W_k A \in S$, there is no equivalent serial schedule s.t. $T_k$ is between $T_i$ and $T_j$.
  But, it can be *before $T_i$ or after $T_j$*.

## POLYGRAPH

For a schedule $S$, the **polygraph** $P(S)$ is a tuple $P(S) = (V, E, F)$, where

1. $V$ is the set of transactions in $S$,
2. $E$ is a set of edges, given by the WR-conflicts in $S$,
3. $F$ is a set of pairs of edges (alternatives):
   for all $i \neq j$ such that $S = \ldots W_i A \ldots R_j A \ldots$ and there is no write action to $A$ between $W_i A$ and $R_j A$, and all $W_k A$ in $S$ where $k \neq i, k \neq j$:

$$(T_k \to T_i, T_j \to T_k) \in F.$$

   (include $start = W_0(all)$ and $end = R_\infty(all)$!)

A graph $(V, E')$ is **compatible** to a polygraph $(V, E, F)$ if $E \subseteq E'$ and $E'$ contains for each alternative exactly one of the edges.

A polygraph $P(S) = (V, E, F)$ is **cycle-free** if there is a cycle-free compatible graph $(V, E')$.

**Theorem 6.4**
*A schedule $S$ is serializable if and only if its polygraph is cycle-free.* □

Note: The test for cycle-freeness of a polygraph is NP-complete.



**Example 6.11**
*Consider again Example 6.10.*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | $T_1$: | | | R(X) | | W(Y) | |
| S: | $T_2$: | | | | R(Y) | | W(Y) |
| | $T_3$: | R(Z) | W(Y) | | | | |

*From (2), there is the edge $(3, 2) \in E$.*

*From (3), consider*

- $W_3(Y)/R_2(Y)$*: For* $W_1(Y)$ $((1, 3), (2, 1))$ *has to be added to* $F$.
- $W_0(X)/R_1(X)$, $W_0(Z)/R_3(Z)$*: there is no* $W(X)$ *and* $W(Z)$*. Do nothing. Note that the original value of* $Y$ *is never read.*
- $W_2(Y)/R_\infty(Y)$*. For* $W_1(Y)$ *and* $W_3(Y)$*, add* $((1, 2), (\infty, 1))$ *and* $((3, 2), (\infty, 3))$ *to* $F$.

*Since edges like $(n, 0)$ (a transaction before start) and $(\infty, n)$ (a transaction after end) do not make sense, the only compatible graphs are*

- *(3,2), (1,3), (1,2), (3,2) (cycle-free), and*
- *(3,2), (2,1), (1,2), (3,2) (cyclic).*

*The first of these gives the equivalent serial schedule $T_1 T_3 T_2$.* □

... and now to the

Proof of Theorem 6.4

We need two Lemmata:

**Lemma 6.1**
*For two equivalent schedules $S$ and $S'$, $P(S) = P(S')$.* □

**Proof:** Follows from equality of the D-Graphs (which are also based on WR-conflicts).

**Lemma 6.2**
*For a serial schedule $S$, $P(S)$ is cycle-free.* □

**Proof:** Construct a graph $G$ that contains an edge $T_i \to T_j$ if and only if $T_i$ is before $T_j$ in $S$. $G$ is cycle-free and compatible to $P(S)$.



**Proof of the theorem**

"⇒": follows immediately from the above lemmata.

"⇐": Consider a cycle-free graph $G$ that is compatible to $P(S)$. Let $S'$ a serial schedule according to a topological sorting of $G$.
We show that $S$ and $S'$ are equivalent, i.e., $DG(S) = DG(S')$.

Assume $DG(S) \neq DG(S')$. Thus, there are actions $W_iA, W_kA, R_jA$ from different transactions such that

- in $S$, $T_j$ reads a value of $A$ that has been written by $T_i$. Thus,
  - the $E$ component of $P(S)$ contains an edge $T_i \to T_j$,
  - The $F$ component of $P(S)$ contains a pair $(T_k \to T_i, T_j \to T_k)$.
- in $S'$, $T_j$ reads a value of $A$ that has been written by $T_k$ .

Because of compatibility, $G$ contains the edge $T_i \to T_j$.
Since $S'$ is serial, it is of the form $S' = \ldots T_i \ldots T_j \ldots$.

Since $T_j$ reads $A$ from $T_k$ in $S'$ (assumption), $T_k$ is be executed later than $T_i$, and before $T_j$.
Thus, $S' = \ldots T_i \ldots T_k \ldots T_j \ldots$.

Since $G$ is cycle-free, there are no edges $T_k \to T_i$ or $T_j \to T_k$.

Then, $G$ cannot be compatible to $P(S)$ (the pair in $F$ is not satisfied).

## 6.5 Scheduling

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

- runtime-scheduling, incremental
- at each timepoint, new transactions can "arrive" and have to be considered

### Different Types of Strategies

- Supervise the schedule, and with the first non-serializable action, kill the transaction ($\rightarrow$ C-graph, timestamps)
- Avoidance strategies: avoid at all that non-serializable schedules can be created ($\rightarrow$ Locking),
- Optimistic Strategies: keep things running even into non-serializable schedules, and check only just before committing a transaction ($\rightarrow$ read-set/write-set).



### Scheduling Strategies

- **Based on the conflict graph:**

  The scheduler maintains the conflict graph of the actions executed so far (partial schedule).

  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 ($action$ will never be conflict-free in this schedule) abort $T$ and all transactions that depend on $T$ (i.e., that have read items that have been written by $T$ before), and remove the corresponding actions from $S$. Restart $T$ later.

  Note: not only the CG must be maintained, but all earlier actions that can still be part of a conflict (i.e., for each tuple, all actions backwards until (including) the most recent write).

Exercise: $S_4$ from Example 6.2.

## Scheduling Strategies (Cont'd)

- **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 $S$. Restart $T$ later (with new timestamp).

  **Implementation:** For any action (read and write) on a data item $V$, the latest timestamp is recorded at $V$ as $Z_r(V)$ or $Z_w(V)$. Then, (*) is checked as $Z_?(T) \geq Z_?(V)$ (set "?" according to conflict matrix), and if an action is executed, then $Z(V)$ is set to $Z(T)$.

- Lock-based strategies: see next section.



## Scheduling Strategies (Cont'd)

- **Optimistic Strategies:**
  (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 $S$.

## 6.6 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 6.12**
*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"):

  granted privilege (for the same object):

| requested privilege | 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 6.13**
*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):*

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

□

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 6.14**
*Consider again Example 6.12:*

*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 6.5**
*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.
($T_1$ is then of the form $\ldots\ UX\ \ldots\ LY\ \ldots$)

**Example 6.15**
*Consider the non-2PL transaction from Example 6.14 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) is possible that has been shown not to be serializable:*

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

□



## Properties of 2PL (Cont'd)

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

**Example 6.16**
*The schedule $S$*

$$S = R_1A\ R_2A\ W_2A\ R_3B\ W_3B\ R_1B\ W_1B$$

*is serializable (equivalent to $T_3\ T_1\ T_2$), but there is no way to add LOCK/UNLOCK actions to $T_1$ that satisfy the 2PL requirement such that $S$ is an admissible schedule.* □

## STRICT 2PL

Consider Schedule $S_3$ from Example 6.2 with 2PL-Locks:

$S_3 = L_1A\ R_1A\ W_1A\ L_1B\ U_1A\ L_2A\ R_2A\ W_2A\ L_2B\ U_2A\ R_1B\ W_1B\ U_1B\ R_2B\ W_2B\ U_2B.$

Consider the case that $T_1$ fails as follows:

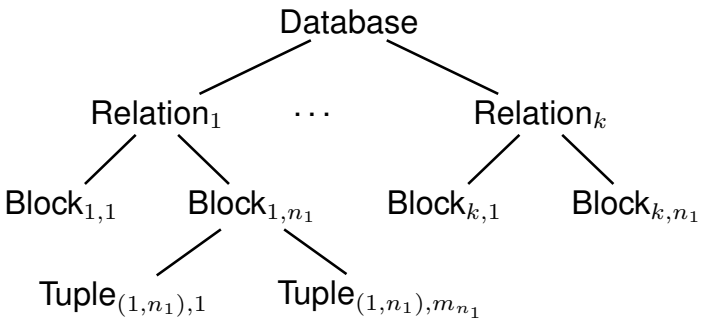
$L_1A\ R_1A\ W_1A\ L_1B\ U_1A\ L_2A\ R_2A\ W_2A\ L_2B\ U_2A\ R_1B\ \text{ROLLBACK}_1$

- $T_2$ has already read $A$ (dirty read) and must also be rolled back.
- Dirty reads (and cascading rollbacks) can be avoided, if the locks are only released *after* EOT ("Strict 2PL"): $T_1 = L_1A\ R_1A\ W_1A\ L_1B\ R_1B\ W_1B\ EOT_1\ U_1A\ U_1B$.
- the user does not have to specify Lock/unlock at all:
  - every item is locked when used for the first time (done via the Access Manager),
  - the transaction manager unlocks all items of a transaction after EOT.



## LOCKING GRANULARITY

- the 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 6.17**
*Consider again Slide 250.*

$T_1$ *computes the sum of the population of all accounts in Frankfurt – reading all these tuples. Thus, at the beginning it locks all (existing) tuples.* $T_2$ *adds a new account and adapts the total.*

*The schedule given on Slide 250 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* EOT (**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)

## 6.7 Safety: Error Recovery

What (more or less 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** ($\Rightarrow$ no cascading rollback).



## "SIMPLE" ROLLBACK

- the transaction manager decides to rollback a running transaction,
- requires to undo all effects of the database
- (recall that strict 2PL is assumed, which avoids dirty reads),
- requires for each transaction a list of what it did.
- these lists could be kept separately for each transaction, or altogether in a "database log" (which will prove useful also in more severe error situations)

## DATABASE LOG

The database system maintains a **log** (also called "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 $WX$:

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

$X_{new}$: value of $X$ written by $T$ **(after image)**

$X_{old}$: value of $X$ before $WX$ **(before image)**

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

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



## TRANSACTION ROLLBACK WITH A DATABASE LOG

Consider the following "money transfer" transaction $T_1$ which additionally sends a confirmation e-mail to $A$.

$T_1 = R_1A(x),\ x = x - 100,\ W_1A(x),\ R_1B(y),\ y = y + 100,\ W_1B(y),\ R_1C(m),$ send mail to $m$.

Given $a_0 = 1000, B_0 = 2000$, and assume that the sending of the mail fails, the log looks as follows:

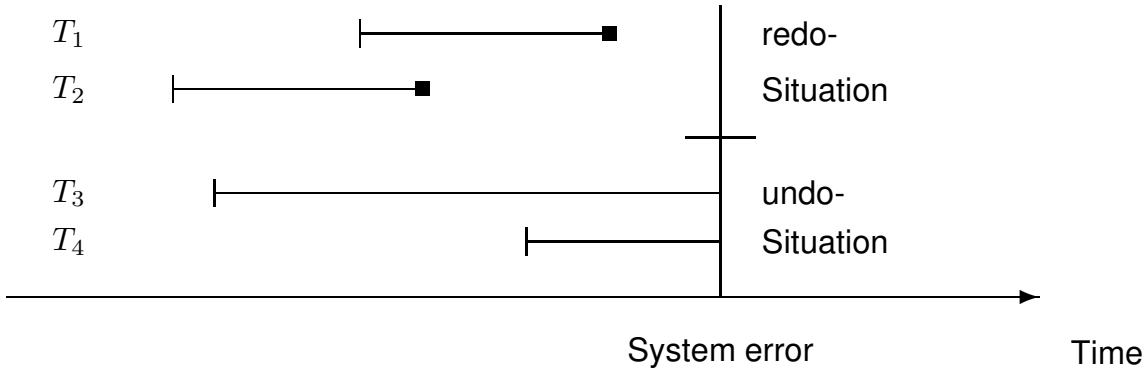
$\ldots (T_1, begin)\ \ldots (T_1, A, 1000, 900)\ \ldots (T_1, B, 2000, 2100) \ldots$

Now, execution of $T_1$ fails when sending the mail.

Scanning the log backwards for entries on $T_1$: set $B$ back to 2000, set $A$ back to 1000, stop going backwards when $(T_1, begin)$ is reached.

(preferable: have an index on the log for each active transaction)

## System errors: 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.



## DB Server Architecture: Secondary Storage and Cache

runtime server system: accessed by user queries/updates

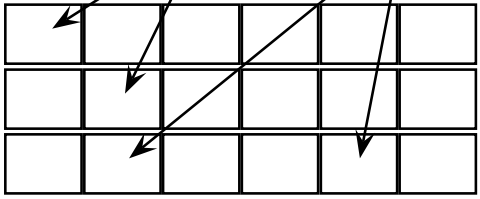
- parser: translates into algebra, determines the required relations + indexes
- file manager: determines the file/page where the requested data is stored
- buffer/cache manager: provides relevant data in the cache
- query/update processing: uses only the cache

Cache (main memory): pagewise organized

- Accessed pages are fetched into the cache
- pages are also changed in the cache
- and written to the database later ...

Secondary Storage (Harddisk): pagewise organized

- data pages with tuples
- index pages with tree indexes (see later)
- database log etc. (see later)



## CACHE VS. MATERIALIZATION IN SECONDARY MEMORY

- operations read and write to cache
- contents of the cache is written ("materialized") in secondary storage at "unknown" timepoints
- if a page is moved out from the cache, its modifications are materialized
- **write immediate:** updates are immediately written to the DB:
  "simple" power failure cannot lead to redo situations; aborted transactions and power failures require to undo materialized updates in the DB.
- write to DB (at latest) **at commit time**.
  then, "simple" power failure can still not lead to redo situations
- **undo-avoiding:**
  write ("materialize") updates to the database **only (at or) after committing**.
  - then, aborted transactions are only concerned with the cache
    (recall that strict 2PL is assumed which prohibits *dirty reads*)
  - any power failure ore media crash cannot lead to undo situations
    (only committed data in DB)



**Example 6.18**
*(write-immediate, no undo-avoiding)*

| | | | | | | |
|---|---|---|---|---|---|---|
| $T_1$: | *BOT LA RA* | | *WA CO UA* | | | |
| $T_2$: | | *BOT LB RB* | | *LA RA WB* | *CO UA UB* | |
| $T_3$: | | | | *BOT LC RC* | | *WC* |
| *Buffers:* | | | | | | |
| $T_1$: | | | $A : f_1(a_0)$ | | | |
| $T_2$: | | | | $B : f_2(f_1(a_0), b_0)$ | | |
| $T_3$: | | | | | | $C : f_3(c_0)$ |
| *Database:* | | | | | | |
| *A:* | $A_0$ | | $f_1(A_0)$ | | | |
| *B:* | $B_0$ | | | $f_2(f_1(a_0), b_0)$ | | |
| *C:* | $c_0$ | | | | | $f_3(c_0)$ |

*Log:*
$(T_1, begin), (T_2, begin), (T_1, A, a_0, f_1(a_0)), (T_1, CO), (T_2, B, b_0, f_2(f_1(a_0), b_0)),$
$(T_3, begin), (T_2, CO), (T_3, C, c_0, f_3(c_0))$ □

## 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.

(Recall 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 6.19**

*Consider again Example 6.18.*

| *(write-immediate, no undo-avoiding)* | | | | *Sys.error* | *state after restart* |
|---|---|---|---|---|---|
| $T_1$: | BOT LA RA | WA CO UA | | | |
| $T_2$: | BOT LB RB | | LA RA WB | CO UA UB | |
| $T_3$: | | | BOT LC RC | WC | |
| *Buffers:* | | | | | |
| $T_1$: | | A : $f_1(a_0)$ | | | |
| $T_2$: | | | B : $f_2(f_1(a_0), b_0)$ | | |
| $T_3$: | | | | C : $f_3(c_0)$ | |
| *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:* | $c_0$ | | | $f_3(c_0)$ | $c_0$ |

*Log:*

$(T_1, begin), (T_2, begin), (T_1, A, a_0, f_1(a_0)), (T_1, CO), (T_2, B, b_0, f_2(f_1(a_0), b_0)),$
$(T_3, begin), (T_2, CO), (T_3, C, c_0, f_3(c_0))$ □



## Logging Requirements

**Log granularity:**

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

**Example 6.20**

*Assume locking at the tuple level, and logging at the relation level, and two transactions:*

$T_1 : \ldots, insert(p(1)), \ldots, eot$
$T_2 : \ldots, insert(p(2)), \ldots, eot$

*and the Schedule* $BOT(T_1), \ldots, T_1 : Lp(1), T_1 : insert(p(1)), BOT(T_2), T_2 : Lp(2), T_2 : insert(p(2)), commit(T_2), \ldots, abort(T_1)$

*The resulting log (initial state of* $p$ *is* $p_0$*) is*

$(T_1, begin), (T_1, p, p_0, p_0 \cup \{1\}), (T_2, begin), (T_2, p, p_0 \cup \{1\}, p_0 \cup \{1\} \cup \{2\}), (T_2, commit)$

*Then the undo operation of* $T_1$ *will erase the result of* $T_2$ *by resetting* $p$ *to* $p_0$*.* □

**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.