# Chapter 4 Parallel Transactions and Execution Control

## Chapter 4.1 Concept of Transaction (see chapter 1.1)

**4.1.1 Syntax:** **begintrans** (BOT)
a1;
a2;
...
ak
**endtrans** (EOT) → **commit** / **abort**

Tansaction $t_j = (a_j1; a_j2; ...; a_jk)$

is a sequential program
$a_jl$ are actions:
r(x) : read Objekt x
w(x) : write Objekt x

**Concept of Object:** single tuple, attribute in tuple, page, relation, Index-page, Index-entry, SQL-Statement (Debit-Credit transaction, see Chapt. 1.1)



- *Atomicity* (Atomarität): all changes of a transaction are performed completely or not at all.
- *Consistency* (Konsistenz): a transaction transforms a consistent database again into a consistent (may be the same) database, i. e., all integrity constraints of the datases are respected by the program.
- *Isolation:* changes of running transactions are not visible for other transactions. A transaction behaves as if there were no other.
- *Durability* (Permanenz): after the successful completion (Commit) of a transaction all updates are permanent, i. e., they survive later errors and failures.

Exchange of information between DB and AS:

1. r(x) → X (main store var.)
2. X → w(x)

## 4.1.2 Interleaving and Anomalies

4 Anomalies:
- Lost Update
- Dirty Read
- Unrepeatable Read
- Phantom

**Example Lost Update**

| $T_1$ | $T_2$ |
|---|---|
| BOT() | |
| r(x) → X | |
| X := X + 1000 | |
| | BOT() |
| | r(x) → Y |
| | Y := X + 5 |
| X → w(x) | |
| Commit() | |
| | Y → w(x) |
| | Commit() |



**Example Dirty Read:**

| $T_1$ | $T_2$ |
|---|---|
| BOT() | |
| r(x) → X | |
| X := X + 1000 | |
| X → w(x) | |
| | BOT() |
| | r(x) → Y |
| Abort() | |
| | Y := Y + 5 |

**Example Unrepeatable Read (inconsistent retrieval)**

| $T_1$ | $T_2$ |
|---|---|
| | BOT() |
| BOT() | |
| | r(y) → Y |
| r(y) → Y | |
| Y := Y - 1000 | |
| Y → w(y) | |
| r(x) → X | |
| X := X + 1000 | |
| X → w(x) | |
| Commit() | |
| | r(x) → X |
| | balance = X - Y |

## Example Phantom

| $T_1$ | $T_2$ |
|---|---|
| | BOT() |
| BOT() | |
| $r(y) \rightarrow Y$ | |
| $r(x) \rightarrow X$ | |
| | $r(y) \rightarrow Y$ |
| | $r(sum) \rightarrow Sum$ |
| | Sum := Sum - Y |
| | delete(y) |
| | $Sum \rightarrow w(sum)$ |
| $r(sum) \rightarrow Z$ | |
| Commit() | |

**Dregrees of Isolation** (consistency levels)

Is. Dg. 3: Repeatable Read (no anomalies)
long read and write locks

Is. Dg. 2: Consistent Read (not repeatable)
short read locks, long write locks

Is. Dg. 1: Browse: no Lost Update
but dirty reads
no readlocks, but long write locks

Is. Dg. 0: Chaos: Lost Updates, but "physically" correct
data structures
only short write locks



## Chapter 4.2 Serializability

**Def.:** - **Schedule for** (History) $t_1 = (a_1 1; a_1 2, ..., a_1 k_1)$
...
$t_e = (a_e 1; ...; a_e k_e)$

H = Sequence of $a_i j$, so that sequence of actions within $t_i$ is preserved (large number of histories)

- **complete Schedule**
  for every $t_i$ $abort_i$ ($ab_i$)
  or $commit_j$ ($c_i$)
  is contained in H

- $t_i$, $t_j$ are **interleaved** in H:
  $a_i l$ **before** $a_j m$ **before** $a_i n$
  we write $a_i l <_H a_j m <_H a_i n$
  i.e.. ... $a_i l$ ... $a_j m$ ... $a_i n$ ... = H

- H is **serial** if no transactions in H are interleaved

**Note:** for l Transactions there are l! serial schedules, many more schedules

**Example:** $t_1 = r_1(x)\ w_1(x)\ c_1$
$t_2 = r_2(x)\ w_2(x)\ c_2$

$H = r_1\ r_2\ w_1\ w_2\ c_1\ c_2$

**Problem:** serial schedules not useful for practical purposes

- slow I/O
- slow process communication
- slow human Interactions

⇒ which interleaved schedules are acceptable?
Equivalence to serial schedule!

**Def.:** State equivalence (final-state equivalence):
schedule H is fs-equivalent to serial schedule $H_s$, if both lead to the same DB-state (semantic equivalence) (too strong)

**Def.:** $a_1i(x)$ and $a_2j(x)$ are in conflict if
$a_1i(x) = w(x)$ or
$a_2j(x) = w(x)$, formally: $\overline{conf}\ (a_1i(x), a_2j(x))$

Conflict relation $\mathbf{conf_H}$ for schedule H:
$\mathbf{conf_H} = \{(a,b) \mid a,b \text{ are in conflict} \wedge a <_H b\}$
we also write $conf_H\ (a,b)$ or a $conf_H$ b.

**Note:** $\overline{conf}$ is symmetric
$conf_H$ is not symmetric



**Example:** $H = r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ c_1\ c_2$
$conf_H = \{(r_1, w_2)$
$(r_2, w_1)$
$(w_1, w_2)\}$

**Def.:** $H_1$, $H_2$ are called **conflict equivalent**,
- if $H_1$, $H_2$ have the same actions and
- $conf_{H_1} = conf_{H_2}$

**Def.:** complete Schedule H is called **conflict serializable** if there exists H':
- H, H' have the same set of actions
- $conf_H = conf_{H'}$
- H' is serial

**Lemma:** conflict serializability
⇒ fs-equivalence (final state equivalence)

**Note:** conflict serializability is the synchronization method of almost all commercial DBS, achievable by synchronization protocol = Concurrency Control Protocol

**Def.:** Interleaved (parallel) execution $\{t_1, ..., t_n\}$ of Trans. $t_1, ..., t_n$ is **correct (consistent)** if:
$\exists i_1, ..., i_n\ [\{t_1, t_2, ..., t_n\} \equiv (ti_1; ti_2; ...; ti_n)]$

many executions (Histories) are possible
specific execution ~ Schedule,
admissible are only conflict-serializable schedules.

## Chapter 4.3 Synchronization

**Def:** Let $H_s$ be serial schedule

$t_1 <^{\bullet} t_2$ in $H_s$, if all actions of $t_1$ are before all actions of $t_2$.

$<^{\bullet}$ is linear order. Denote $H_s = H_{<^{\bullet}}$

**Note:** For a conflict serializable schedule H only conflicting actions are relevant.

(1) as soon as $\exists\, a_i m\ \exists\, a_j n : conf\,(a_i m, a_j n)$ in H
we must have

(2) $\forall\, a_i m\ \forall a_j n : \overline{conf}\,(a_i m, a_j n) \supset a_i m$ **before** $a_j n$
resp. ... $\overline{conf}\,(...) \supset conf\,(...)$

How can (1) $\Rightarrow$ (2) be achieved?

### Strategies for Serialization

1. Serialization according to the sequence of transaction calls

2. Parallelism (interleaving) only between conflict free transactions, i.e.
    - all objects needed by t must be known in advance
    - check conflicts with running t'



- start t only, if there are no conflicts

$\Rightarrow$ all locks of t must be placed before starting t, i.e. $t \;^{\bullet}\!\!> t'$;
no deadlock problem

3. Dynamic acquisition of lock, as soon as t wants to process object o.
    - 3.1 lock granted $\Rightarrow$ t continues
    - 3.2 lock not granted, i.e. o is locked by t' with incompatible lock.
      $\Rightarrow$ t must wait for t'

### Deadlock - Problem:

**Def.:** Follow - Relation $\rightarrow$ for transactions in H:

$t_j \rightarrow t_i$ if (1)

i. e. $\exists a_i m \; \exists a_j n$ : conf $(a_i m, a_j n)$

and $t_j$ follows $t_i$

**Lemma:** H is conflict serializable, if $\rightarrow$ is acyclic

**Proof:** Construct topological order $<^{\bullet}$ corresponding to $\rightarrow$H

$<^{\bullet}$ is conflict equivalent to H

## Basic Idea: Object locks

Compute conf $(a_i l, a_j k)$ and simultaneously

$\rightarrow$ incrementally with the aid of locks on objects

reason: conflicts arise because of access to objects

$r_i(o); r_j(o)$ : no conflict
compatible locks

$r_i(o); r_j(o)$ : $t_i$ places R-lock on o

$w_j(o); r_j(o)$ : $t_i$ places X-lock on o

$w_i(o); w_j(0)$ : $t_i$ places X-lock on o

update $\rightarrow$ and check for acyclicity $t_j \rightarrow t_i$
means $t_j$ waits for $t_i$



## Lock Compatibility

| | R | X |
|---|---|---|
| R | | |
| X | | |

## 2-Phase Lock Protocol

Example:

1. $t_1, t_2, t_3$ request R-lock on o1
2. $t_4$ requests X-lock on o2
3. $t_5$ requests X-lock on o1
4. $t_6$ requests R-lock on o2
5. $t_7$ requests R-lock on o1

Passing of transactions?
Starvation?

**Note:** Locks alone do not guarantee serialization (consistency)

$t_1$ : A := 1; B := 0 $\quad$ $t_2$ : B := 1; A := 0

Special interleaving (parallelism) with sequence w.r. to time:

| | | |
|---|---|---|
| $\{t_1, t_2\}^* \equiv$ | $A_1$ := 1; | ($t_1$, A, X) |
| | $B_2$ := 1; | A := 1 |
| | $B_1$ := 0; | release ($t_1$, A, X) |
| | $A_2$ := 0 | . . . |

$\Rightarrow A = 0 \wedge B = 0$

but : $(t_1; t_2) \Rightarrow B = 1 \wedge A = 0$

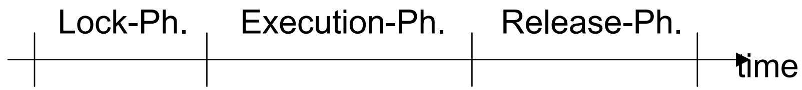
$(t_2; t_2) \Rightarrow A = 1 \wedge B = 0$

$(t_1; t_2) \not\equiv \{t_1, t_2\}^* \not\equiv (t_2; t_1)$

i.e. $\{t_1, t_2\}^*$ is not conflict-serializable (no consistent parallel execution)

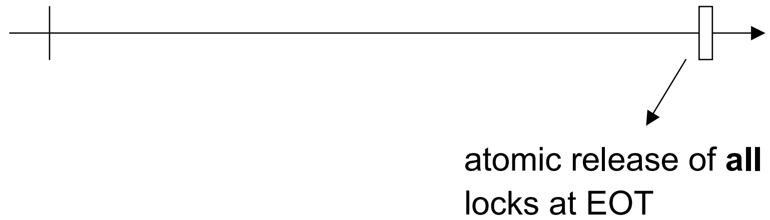


**2 Phases w.r. to Locks:**



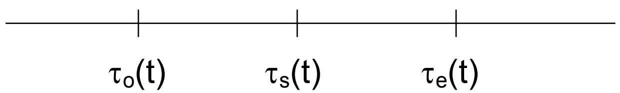


**Strict 2-Ph. Lock-Protocol:**




**Theorem:** 2-phase lock protocol results in conflict-serializable schedules (consistent execution control)

**Proof:**



$\tau_s(t)$ : time, when last lock is granted

$\tau_s(t)$ : t has locked all needed objects, may continue without inhibition, no further waiting!

$\tau_s(t_1) < \tau_s(t_2)$ : $t_1$ in serialization before $t_2$ : $t_2 \to t_1$

$\cdot\!>$

Assumption: $t_1$ wants to lock $\{o_1, ..., o_n\} =: O$
$t_2$ wants to lock $\{p_1, ..., p_m\} =: P$

**Case 1:** $O \cap P = \emptyset$

$\{t_1, t_2\} \equiv (t_1; t_2) \equiv (t_2; t_1)$
no conflicts!

**Case 2:** $O \cap P \neq \emptyset$

let $O_j = P_k \in O \cap P$ locked simultaneously by $t_1$, $t_2$, only possible for R-locks $\Rightarrow o_j$ only read by $t_1$, $t_2$,
$(t_1; t_2) \equiv (t_2; t_1)$

**Case 3:** $O \cap P \neq \emptyset$

let $o_j = p_k$ with conflicting locks: "who locks first, finishes first"

e.g. $t_2 \to t_1$ w.r. to $o_j$

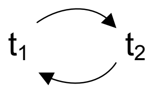
Let $O_K \subseteq O \cap P$; $O_K$ set of objects requested by $t_1$, $t_2$ with conflicting locks:



a) $t_1$ locks all of $O_K$ first :
$t_2 \to t_1$ and $t_1 <\!\bullet\; t_2$

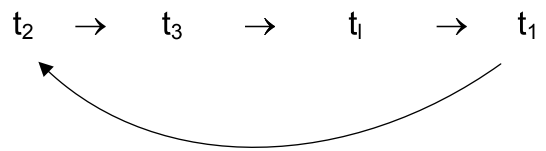
b) $t_2$ locks all of $O_K$ first:
$t_1 \to t_2$ and $t_2 <\!\bullet\; t_1$

c) $t_1$ locks $O_1 \subset O_K$ and $O_1 \neq \emptyset$
$\neq$
$t_2$ locks $O_2 \subset O_K$ and $O_2 \neq \emptyset$
$\neq$
then $t_1 \to t_2$ to free $O_2$
$t_2 \to t_1$ to free $O_1$

i. e. 


d) general case: $O_{Ki,j}$ = set of objects needed by $t_i$ and $t_j$ with conflicts.

$O_{K1,2}$ $O_{K2,3}$ ... $O_{Kl-1,l}$ $O_{Kl,1}$

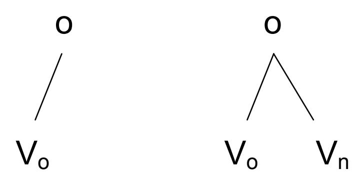



**Note:** 2-Phase protocol determines sequence if conflicts arise locally: t1 → t2 or t2 → t1

globally: wait graph and detection of cylces
= deadlock

**Note:** Sink of the wait graph and isolated nodes
= transactions in process state "ready"
(rechenwillig)

**Example:**




1. $t_1$ updates o to $V_n^1$, frees o
2. $t_2$, $t_3$ read $V_n^1$
3. $t_4$ reads $V_n^1$, changes $V_n^1$ to $V_n^2$
4. $t_1$ fails, is aborted,
   e.g. because of diskfailure, integrity violation

⇒ $t_2$, $t_3$, $t_4$ must be aborted, cascading!!

**Theorem:** Strict 2-Phase locking protocol avoids cascading when transactions are aborted.

**Proof:** reason for cascading:
versions of objects were read, which were not yet committed, since locks (and objects) were freed before commit.

Example with strict protocol?