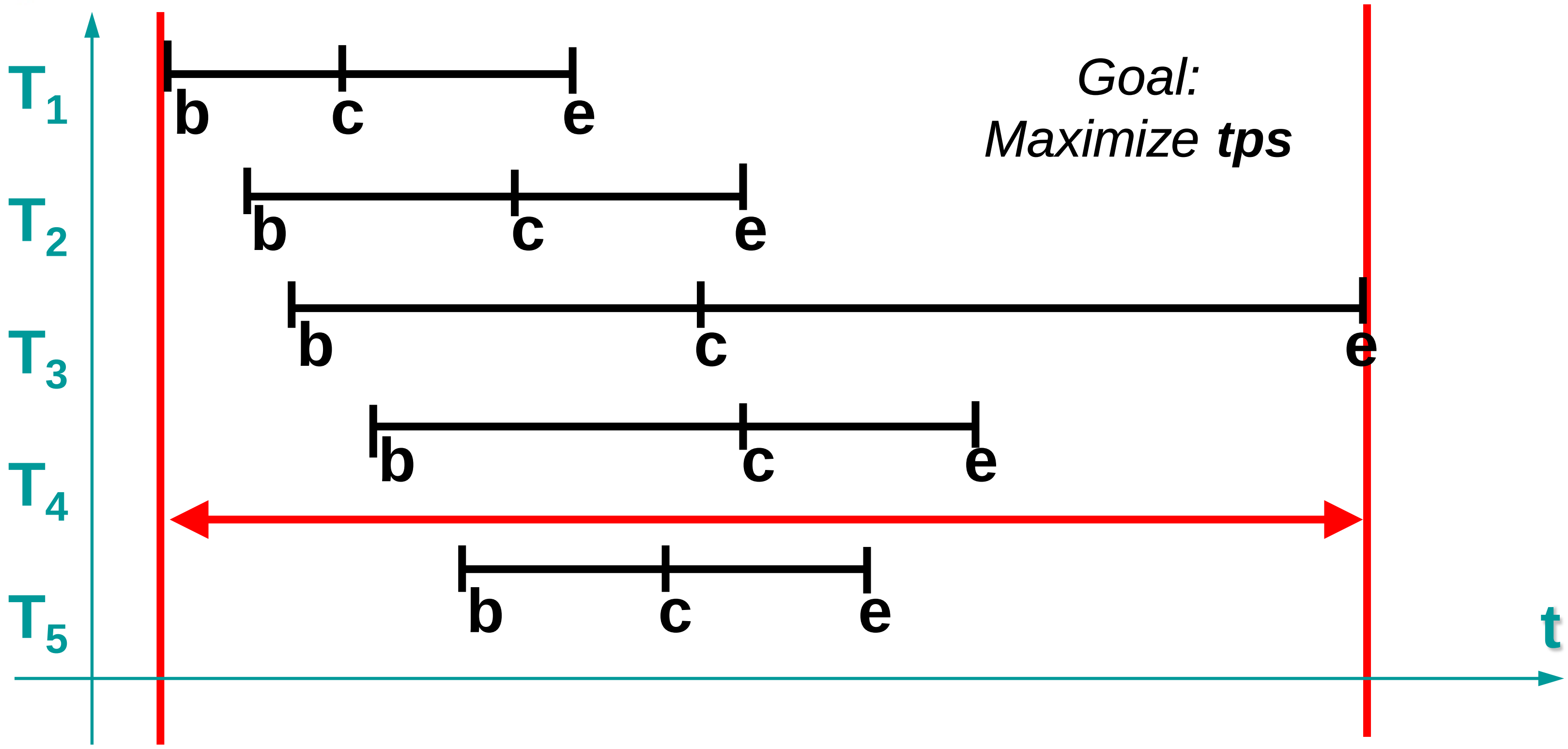
Databases 2

2 Concurrency Control

Advantages of Concurrency



Problems due to Concurrency

T_1:
```
begin transaction
update account
 set balance = balance + 3
 where customer = 'Smith'
commit work
end transaction
```

T_2:
```
begin transaction
update account
 set balance = balance + 6
 where customer = 'Smith'
commit work
end transaction
```

Concurrent SQL transactional statements addressing the same resource

T_1 :
```
begin transaction
D = D + 3
commit work
end transaction
```

T_2 :
```
begin transaction
D = D + 6
commit work
end transaction
```

Lower level view of the transactions

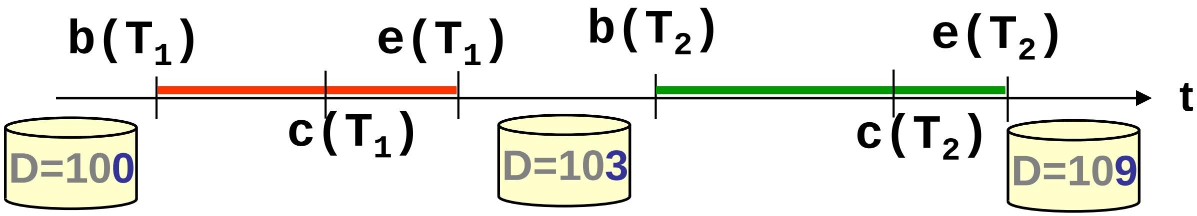
T_1 :
```
begin transaction
read(D,x)
x = x + 3
write(x,D)
commit work
end transaction
```

T_2 :
```
begin transaction
read(D,y)
y = y + 6
write(y,D)
commit work
end transaction
```

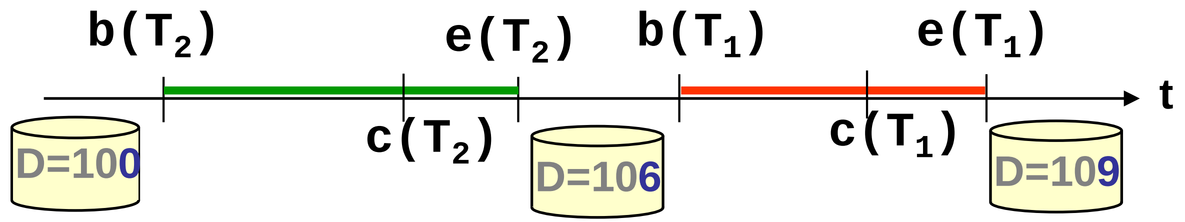
Serial Executions

```
bot(T1)
 r(D,x)
 x = x + 3
 w(x,D)
commit(T1)
eot(T1)
```

Initially D_0=100




```
bot(T2)
 r(D,y)
 y = y + 6
 w(y,D)
commit(T2)
eot(T2)
```

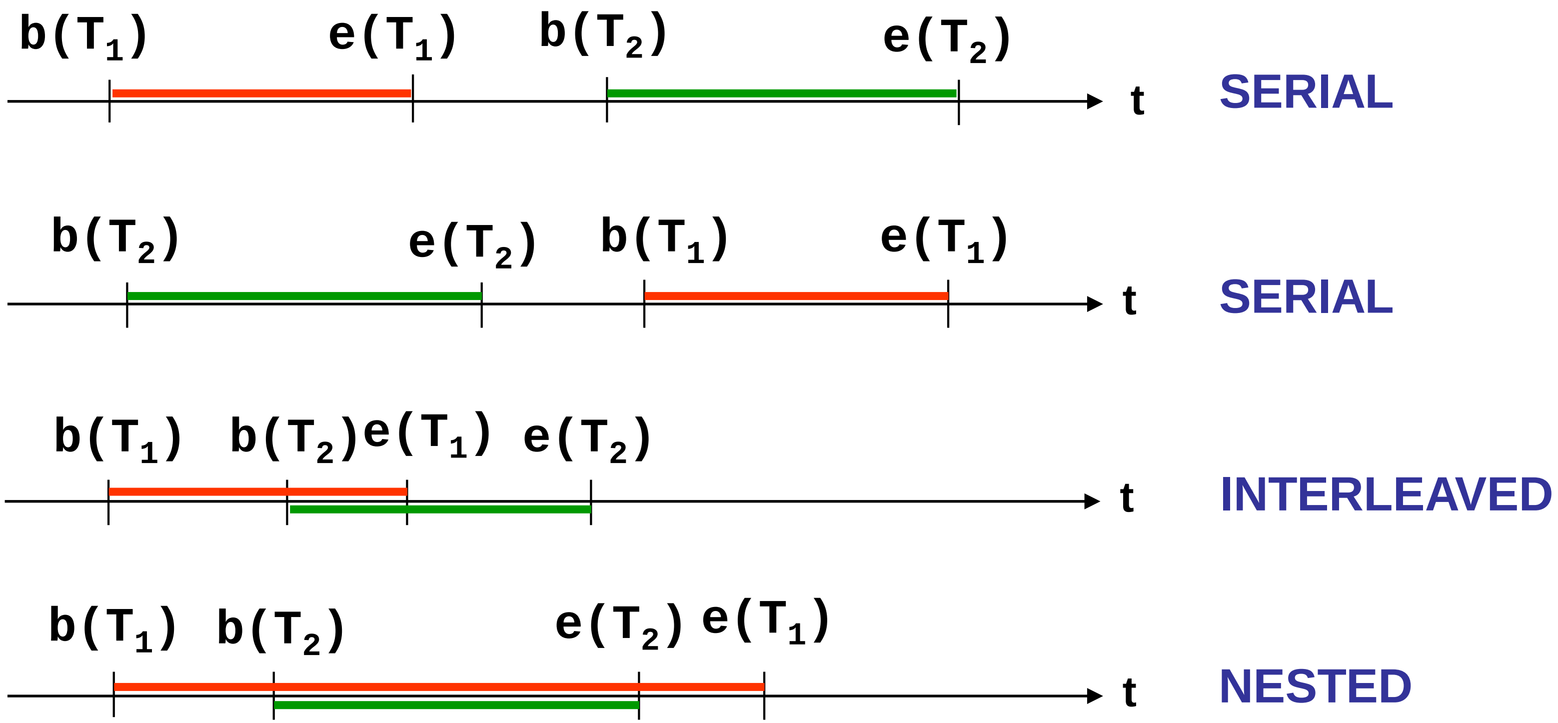


Concurrency Control

- Concurrency is fundamental
  - Tens or hundreds of transactions per second cannot be executed serially
- Examples: banks, ticket reservations

- **Problem**: concurrent execution may cause **anomalies**
  - Concurrency needs to be controlled

Concurrent Executions



Execution with Lost Update

```
bot(T1)
    r(D,x)        D=100
    x=x+3
                          bot(T2)
                              r(D,y)   D=100
   w(x,D)         D=103
   commit
eot(T1)
                              y=y+6
                              w(y,D)    D=106
                              commit
                          eot(T2)
```

T1 : **UPDATE account**
SET balance = balance + 3
WHERE client = 'Smith'

T2 : **UPDATE account**
SET balance = balance + 6
WHERE client = 'Smith'

Sequence of I / O Actions producing the Error

r_1 r_2 w_1 w_2

or

r_1 r_2 w_2 w_1

"Dirty" Read

D=100

```
bot(T1)
    r(D,x)    D=100
    x=x+3
    w(x,D)    D=103
                          bot(T2)
                              r(D,y)   D=103
    ROLLBACK;
eot(T1)       D=100
                              y=y+6
                              w(y,D)   D=109
                              commit
                          eot(T2)
```

T1 : **UPDATE account**
SET balance = balance + 3
WHERE client = 'Smith'

T2 : **UPDATE account**
SET balance = balance + 6
WHERE client = 'Smith'

"Nonrepeatable" Read

D=100

```
bot(T1)
    r(D,x)   D=100
                        bot(T2)
                            r(D,y)   D=100
                            y=y+6
                            w(y,D)   D=106
                            commit
                        eot(T2)
    r(D,z)   D=106
    ...
```

T1 : **SELECT balance FROM account**
WHERE client = 'Smith'
UPDATE account
SET balance = balance + 3
WHERE client = 'Smith'

T2 : **UPDATE account**
SET balance = balance + 6
WHERE client = 'Smith'

Phantom Update

X+Y+Z=100, X=50, Y=30, Z=20

```
bot(T1)
   r(X,V1)
   r(Y,V2)
                    bot(T2)
                       r(Y,V3)
                       r(Z,V4)
                       V3=V3+10
                       V4=V4-10
                       w(V3,Y)
                       w(V4,Z)
    r(Z,V5)
                       ...
```

(Y=40, Z=10)

(but for T1, X+Y+Z=90!)

Phantom Insert

```
bot(T1)
  c=avg(X:A=1)
                  bot(T2)
                    insert X(A=1,B=2)
  c=avg(X:A=1)
eot(T1)
                     ...
                  eot(T2)
```

- Note: this anomaly does not depend on data already present in the DB when T1 executes, but on a “phantom” tuple that is inserted and satisfies the conditions of a previous query

Anomalies

Lost update	r_1 - r_2 - w_2 - w_1
Dirty read	r_1 - w_1 - r_2 - abort_1 - w_2
Nonrepeatable read	r_1 - r_2 - w_2 - r_1
Phantom update	r_1 - r_2 - w_2 - r_1
Phantom insert	r_1 - w_2(*new data*) - r_1

Schedule

- Sequence of input/output operations performed by concurrent transactions

T_1 : $r_1(x)$ $w_1(x)$

T_2 : $r_2(z)$ $w_2(z)$

S_1: $r_1(x)$ $r_2(z)$ $w_1(x)$ $w_2(z)$

Schedules

- How many distinct schedules exist for two transactions?
  - With T_1 and T_2 from the previous slide:

$$\left.\begin{array}{l} r_1(x)\ w_1(x)\ r_2(z)\ w_2(z) \\ r_2(z)\ w_2(z)\ r_1(x)\ w_1(x) \end{array}\right\} \textit{serial}$$

$$\left.\begin{array}{l} r_1(x)\ r_2(z)\ w_1(x)\ w_2(z) \\ r_2(z)\ r_1(x)\ w_2(z)\ w_1(x) \end{array}\right\} \textit{interleaved}$$

$$\left.\begin{array}{l} r_1(x)\ r_2(z)\ w_2(z)\ w_1(x) \\ r_2(z)\ r_1(x)\ w_1(x)\ w_2(z) \end{array}\right\} \textit{nested}$$

$$N = 6$$

Schedules

- How many distinct schedules exist for n transactions? N_D
- How many of them are serial? N_S
- n transactions $(T_1, T_2, \ldots, T_i, \ldots, T_n)$, each with k_i operations

  T_1 has k_1 operations, T_i has k_i operations...

T_1: $o_1^1\ o_1^2\ \ldots\ o_1^{k_1}$

T_2: $o_2^1\ o_2^2\ \ldots\ o_2^{k_2}$

...

T_i: $o_i^1\ o_i^2\ \ldots\ o_i^{k_i}$

...

T_n: $o_n^1\ o_n^2\ \ldots\ o_n^{k_n}$

$$N_S = n!$$

$$N_D = \frac{\left(\sum_{i=1}^{n} k_i\right)!}{\prod_{i=1}^{n}(k_i!)}$$

$$N_S << N_D$$

Principles of Concurrency Control

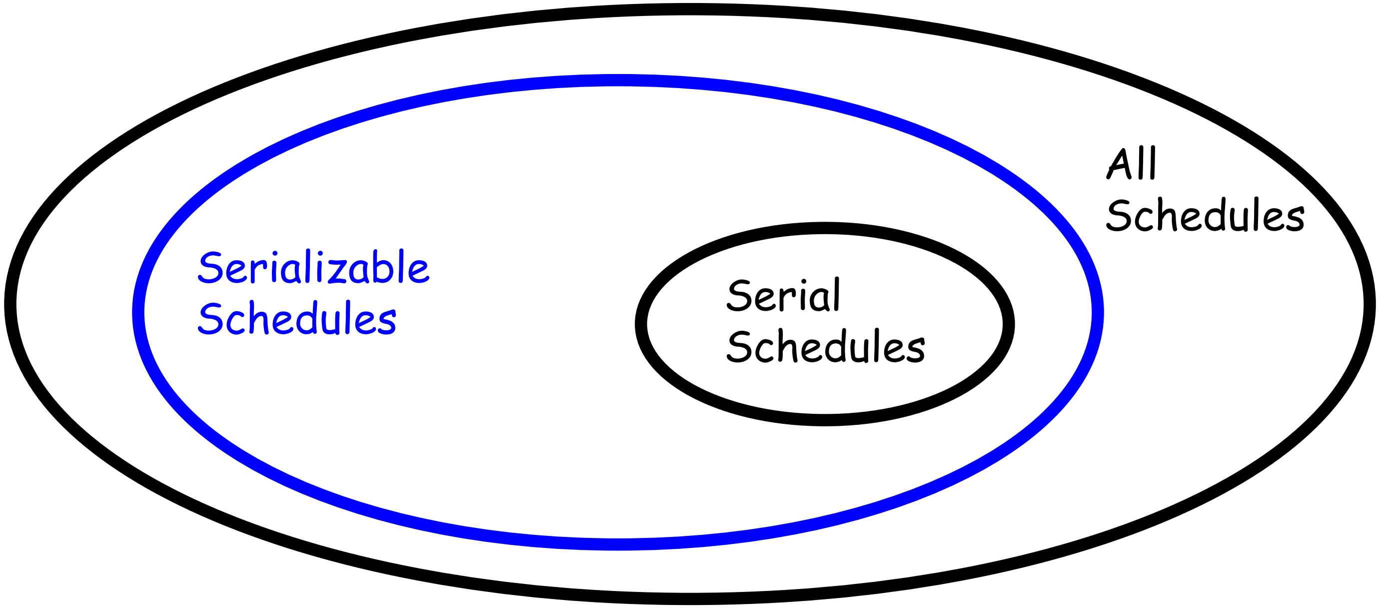
- **Goal**: to reject schedules that cause anomalies
- ***Scheduler***: a component that accepts or rejects the operations requested by the transactions
- ***Serial schedule***: a schedule in which the actions of each transaction occur in a contiguous sequence

S_2: $r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$

Principles of Concurrency Control

- ***Serializable schedule***
  - A schedule that produces the ***same results*** as some serial schedule of the same transactions
  - Requires a notion of schedule equivalence
    - Different notions → different classes (cost of checking)
- Assumption
  - We initially assume that transactions are observed "in the past" (***commit-projection***), and we decide whether the corresponding schedule is correct
  - In practice (and in contrast), schedulers must take decisions ***while transactions are running***

Basic Idea



View-serializability

- Preliminary definitions:
  - $r_i(x)$ ***reads-from*** $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ and there is no $w_k(x)$ in S between $r_i(x)$ and $w_j(x)$
  - $w_i(x)$ in a schedule S is a ***final write*** if it is the last write on x that occurs in S
- Two schedules are ***view-equivalent*** ($S_i \approx_V S_j$) if: they have the same <u>operations</u>, the same <u>reads-from</u> relation, and the same <u>final writes</u>
- A schedule is ***view-serializable*** if it is view-equivalent to a serial schedule of the same transactions
- The class of view-serializable schedules is named **VSR**

Examples of View-serializability

S_3: $w_0(x)$ $\underline{r_2(x)}$ $r_1(x)$ $w_2(x)$ $w_2(z)$

S_4: $w_0(x)$ $r_1(x)$ $r_2(x)$ $w_2(x)$ $w_2(z)$

serial

S_5: $w_0(x)$ $r_1(x)$ $w_1(x)$ $r_2(x)$ $\underline{w_1(z)}$

S_6: $w_0(x)$ $r_1(x)$ $w_1(x)$ $w_1(z)$ $r_2(x)$

- S_3 is view-equivalent to serial schedule S_4 (so it is view-serializable)
- S_5 is not view-equivalent to S_4, but it is view-equivalent to serial schedule S_6, so it is also view-serializable

Examples of View-serializability

S_7 : $r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$

S_8 : $r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$

S_9 : $r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$

- S_7 corresponds to a lost update
- S_8 corresponds to a non-repeatable read
- S_9 corresponds to a phantom update
- They are all **non** view-serializable

A More Complex Example

S_{10} : $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $w_0(y)$ $r_3(z)$ $w_3(z)$ $w_2(y)$ $w_1(x)$ $w_3(y)$

Is S_{10} serializable?

Yes iff there exists a serial schedule S_s s.t. $S_{10} \approx_{\mathbf{V}} S_s$

S_{11} : T_0 T_1 T_2 T_3 is **not** view equivalent to S_{10}

$w_0(x)$ $w_0(z)$ $w_0(y)$ $r_1(x)$ $r_1(z)$ $w_1(x)$ $r_2(x)$ $w_2(y)$ $r_3(z)$ $w_3(z)$ $w_3(y)$

What do we conclude?

Let's try with S_{12} : T_0 T_2 T_1 T_3

$w_0(x)$ $w_0(z)$ $w_0(y)$ $r_2(x)$, $w_2(y)$ $r_1(x)$ $r_1(z)$ $w_1(x)$ $r_3(z)$ $w_3(z)$ $w_3(y)$

A More Complex Example

S_{10}: $w_0(x)\ r_1(x)\ w_0(z)\ r_1(z)\ r_2(x)\ w_0(y)\ r_3(z)\ w_3(z)\ w_2(y)\ w_1(x)\ w_3(y)$

S_{12}: $w_0(x)\ w_0(z)\ w_0(y)\ \ r_2(x)\ w_2(y)\ \ r_1(x)\ r_1(z)\ w_1(x)\ \ r_3(z)\ w_3(z)\ w_3(y)$

A More Complex Example

S_{10}: $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $w_0(y)$ $r_3(z)$ $w_3(z)$ $w_2(y)$ $w_1(x)$ $w_3(y)$

S_{12}: $w_0(x)$ $w_0(z)$ $w_0(y)$ $r_2(x)$ $w_2(y)$ $r_1(x)$ $r_1(z)$ $w_1(x)$ $r_3(z)$ $w_3(z)$ $w_3(y)$

reads-from OK: $r_1(x)$ from $w_0(x)$,
$r_1(z)$ from $w_0(z)$,
$r_2(x)$ from $w_0(x)$,
$r_3(z)$ from $w_0(z)$,

final writes OK: $w_1(x)$, $w_3(y)$, $w_3(z)$

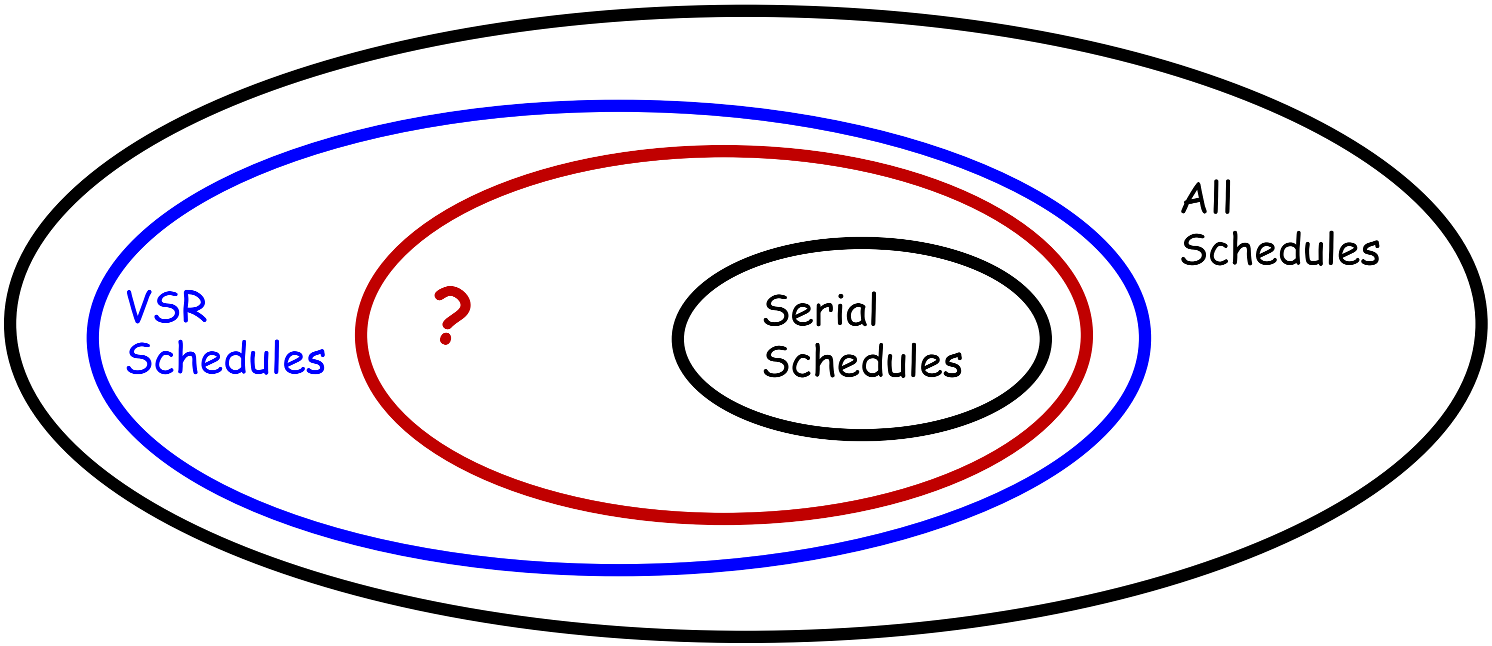
$S_{10} \in VSR$

Complexity?

Complexity of View-serializability

- Deciding view-equivalence of **two given** schedules is done in **polynomial** time and space
- Deciding view-serializability of a **generic** schedule is an **NP-complete** problem
  - *requires considering the reads-from and final writes of* ***all possible*** *serial schedules with the same operations –* ***combinatorial*** *in the general case*
  - ***OMG, performance****!! : what can we trade for that?*
    - *…Accuracy!*

VSR schedules are "too many"



Conflict-serializability

- Preliminary definition:
  - Two operations o_i and o_j ($i \neq j$) are in **conflict** if they address the same resource and at least one of them is a write
    - *read-write* conflicts (*r-w* or *w-r*)
    - *write-write* conflicts (*w-w*)

Conflict-serializability

- Two schedules are **conflict-equivalent** ($S_i \approx_C S_j$) if :
  S_i and S_j contain the same operations and
  all conflicting pairs occur in the same order
- A schedule is **conflict-serializable** if it is conflict-equivalent to a serial schedule of the same transactions
- The class of conflict-serializable schedules is named **CSR**

Relationship between CSR and VSR

- **VSR $\supset$ CSR** : all conflict-serializable schedules are also view-serializable, but the converse is not necessarily true. Proofs:
- Counter-example: we consider $r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$ that
  - is view-serializable: it is view-equivalent to

    $T_1T_2T_3 = r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$
  - is not conflict-serializable, due to the presence of

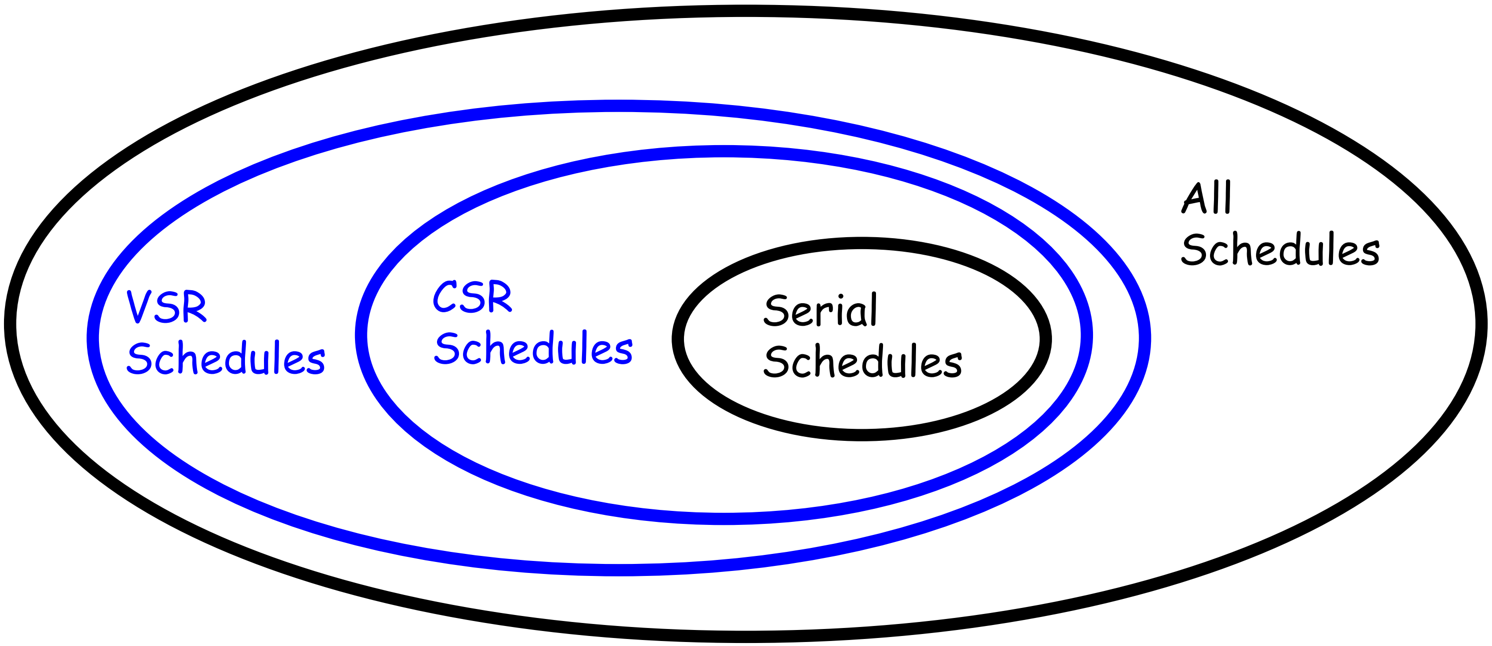
    $r_1(x)\ w_2(x)$ and $w_2(x)\ w_1(x)$

    there is no conflict-equivalent serial schedule, neither $T_1T_2T_3$ nor $T_2T_1T_3$

CSR implies VSR

- **CSR ➔ VSR**: conflict-equivalence $\approx_C$ implies view-equivalence $\approx_V$
- We assume $S_1 \approx_C S_2$ and prove that $S_1 \approx_V S_2$. S_1 and S_2 have:
  - The same final writes: if they didn't, there would be at least two writes in a different order, and since two writes are conflicting operations, the schedules would not be $\approx_C$
  - The same "reads-from" relations: if not, there would be at least one pair of conflicting operations in a different order, and therefore, again, $\approx_C$ would be violated

CSR and VSR


All
Schedules
VSR
Schedules
CSR
Schedules
Serial
Schedules

Testing conflict-serializability

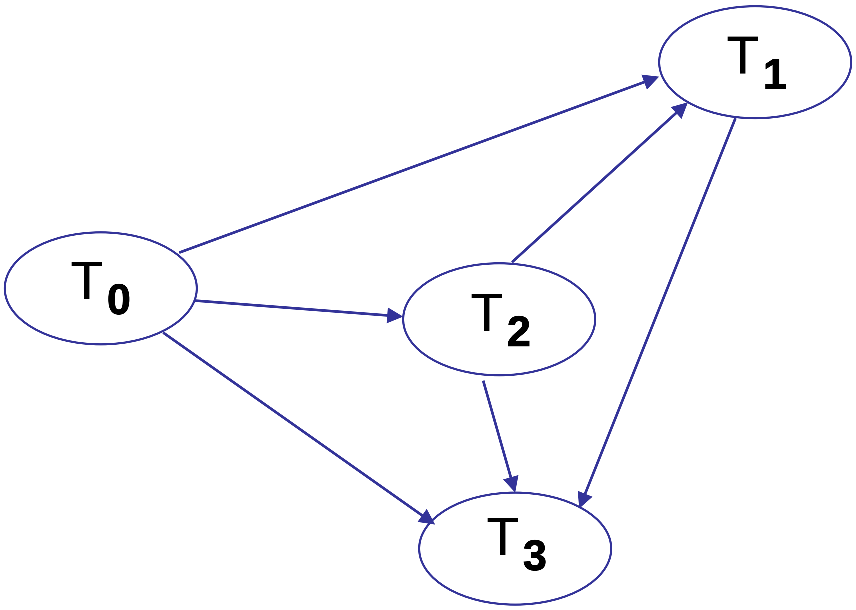
- Is done with a *conflict graph* that has:
  - One node for each transaction T_i
  - One arc from T_i to T_j if there exists at least one conflict between an operation o_i of T_i and an operation o_j of T_j such that o_i precedes o_j
- Theorem:
  - A schedule is in CSR if and only if its conflict graph is acyclic

Testing conflict-serializability

S_{10} : $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $w_0(y)$ $r_3(z)$ $w_3(z)$ $w_2(y)$ $w_1(x)$ $w_3(y)$

resource-based projections:

- **x** : w_0 r_1 r_2 w_1
- **y** : w_0 w_2 w_3
- **z** : w_0 r_1 r_3 w_3



CSR implies Acyclicity of the Conflict Graph

- Consider a schedule S in CSR. As such, it is $\approx_C$ to a serial schedule
- W.l.o.g. we can (re)label the transactions of S to say that their order in the serial schedule is: T_1 T_2 ... T_n
- Since the serial schedule has all conflicting pairs in the same order as schedule S, in the conflict graph there can only be arcs (i,j), with i<j
- Then the graph is acyclic, as a cycle requires at least an arc (i,j) with i>j

Acyclicity of the Conflict Graph implies CSR

- If S's graph is acyclic then it induces a *topological (partial) ordering* on its nodes, i.e., an ordering such that the graph only contains arcs (i,j) with i<j. The same partial order exists on the transactions of S.
- Any serial schedule whose transactions are ordered according to the partial order is conflict-equivalent to S, because for all conflicting pairs (i,j) it is always i<j
  - In the example before: $T_0 < T_2 < T_1 < T_3$
  - In general, there can be **many** compatible serial schedules (i.e., many serializations for the same acyclic graph)

Concurrency Control in Practice

- This technique would be efficient if we knew the graph from the beginning — but we don't
  - A scheduler must rather work "**incrementally**", i.e., decide for each requested operation whether to execute it immediately or to reject/delay it
  - It is not feasible to maintain the conflict graph, update it, and check its acyclicity at each operation request
  - Some simpler, on-line "decision criterion" is required for the scheduler to have **negligible overhead**

Locking

- It's the most common method in commercial systems
- A transaction is well-formed w.r.t. locking if
  - **`read`** operations are preceded by **`r_lock`** (SHARED LOCK) and followed by **`unlock`**
  - **`write`** operations are preceded by **`w_lock`** (EXCLUSIVE LOCK) and followed by **`unlock`**
- Transactions that first read and then write an object may:
  - Acquire a **`w_lock`** already when reading
  - Upgrade a **`r_lock`** into a **`w_lock`** (lock escalation)

Lock Primitives

- Possible states of an object:
  - **`free`**
  - **`r-locked`** (locked by one or more readers)
  - **`w-locked`** (locked by a writer)
- Primitives:
  - **`r-lock`**: read lock
  - **`w-lock`**: write lock
  - **`unlock`**

Behavior of the Lock Manager (Conflict Table)

- The lock manager receives the primitives from the transactions and grants resources according to the conflict table
  - When a **`lock`** request is granted, the resource is acquired
  - When an **`unlock`** is executed, the resource becomes available

REQUEST	RESOURCE STATUS		
	FREE	R_LOCKED	W_LOCKED
`r_lock`	OK R_LOCKED	OK R_LOCKED (n++)	NO W_LOCKED
`w_lock`	OK W_LOCKED	NO R_LOCKED	NO W_LOCKED
`unlock`	*ERROR*	OK *DEPENDS* (n--)	OK FREE

n: counter of the concurrent readers, (inc|dec)remented at each (`r_`|`un`)`lock`

Example

$r_1(x)$ r_1-lock(x) request → OK → x state = r-locked, r=1

$w_1(x)$ w_1-lock(x) request → OK (upgrade) → x state = w-locked

$r_2(x)$ r_2-lock(x) request → NO because w-locked → T2 waits

$r_3(y)$ r_3-lock(y) request → OK → y state = r-locked

T3 releases its locks → y state = free

$w_1(y)$ w_1-lock(y) request → OK → y state = w-locked

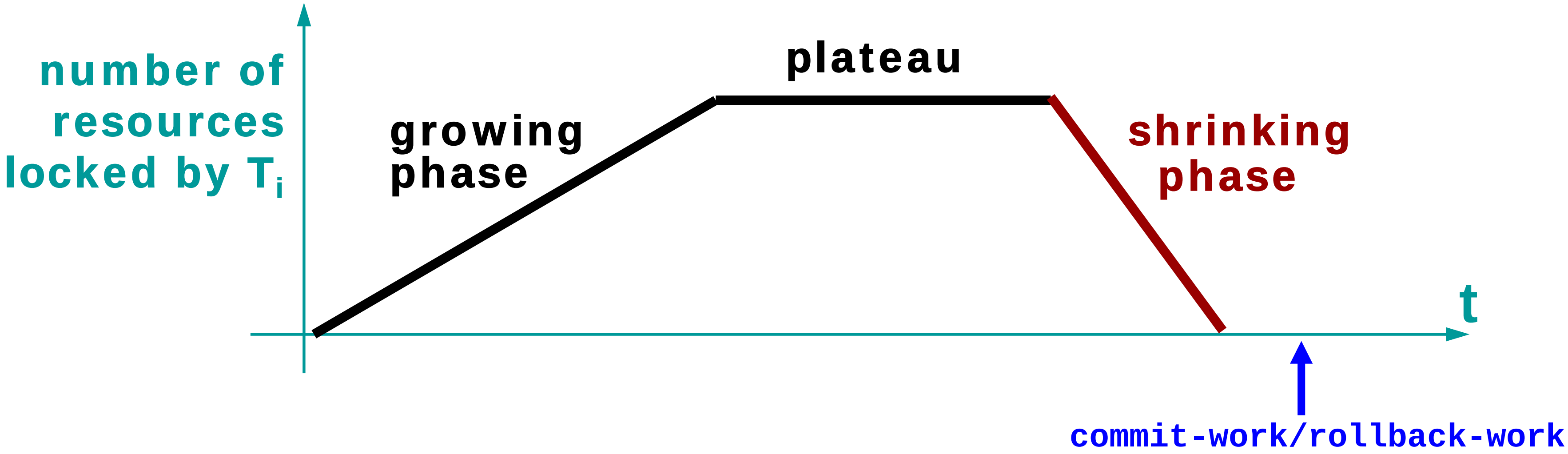
T1 releases its locks → x and y states = free

$r_2(x)$ was waiting for r_2-lock(x) request → OK x state = r-locked

....

Two-Phase Locking (2PL)

- Requirement (two-phase rule):
  - A transaction cannot acquire any other lock after releasing a lock



Serializability

- Consider a scheduler that
  - Only processes well-formed transactions
  - Grants locks according to the conflict table
  - Checks that all transactions apply the two-phase rule
- The class of generated schedules is called **2PL**

Schedules in 2PL are view- and conflict-serializable

(VSR $\supset$ CSR $\supset$ 2PL)

2PL and CSR

- **CSR ⊃ 2PL:** Every 2PL schedule is also conflict-serializable, but the converse is not necessarily true
- Counter-example: $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$
  - It violates 2PL

    $r_1(x)\ w_1(x)\ |\ r_2(x)\ w_2(x)\ r_3(y)\ |\ w_1(y)$

    T_1 releases **T_1 acquires**
  - However, it is conflict-serializable

    $T_3 < T_1 < T_2$

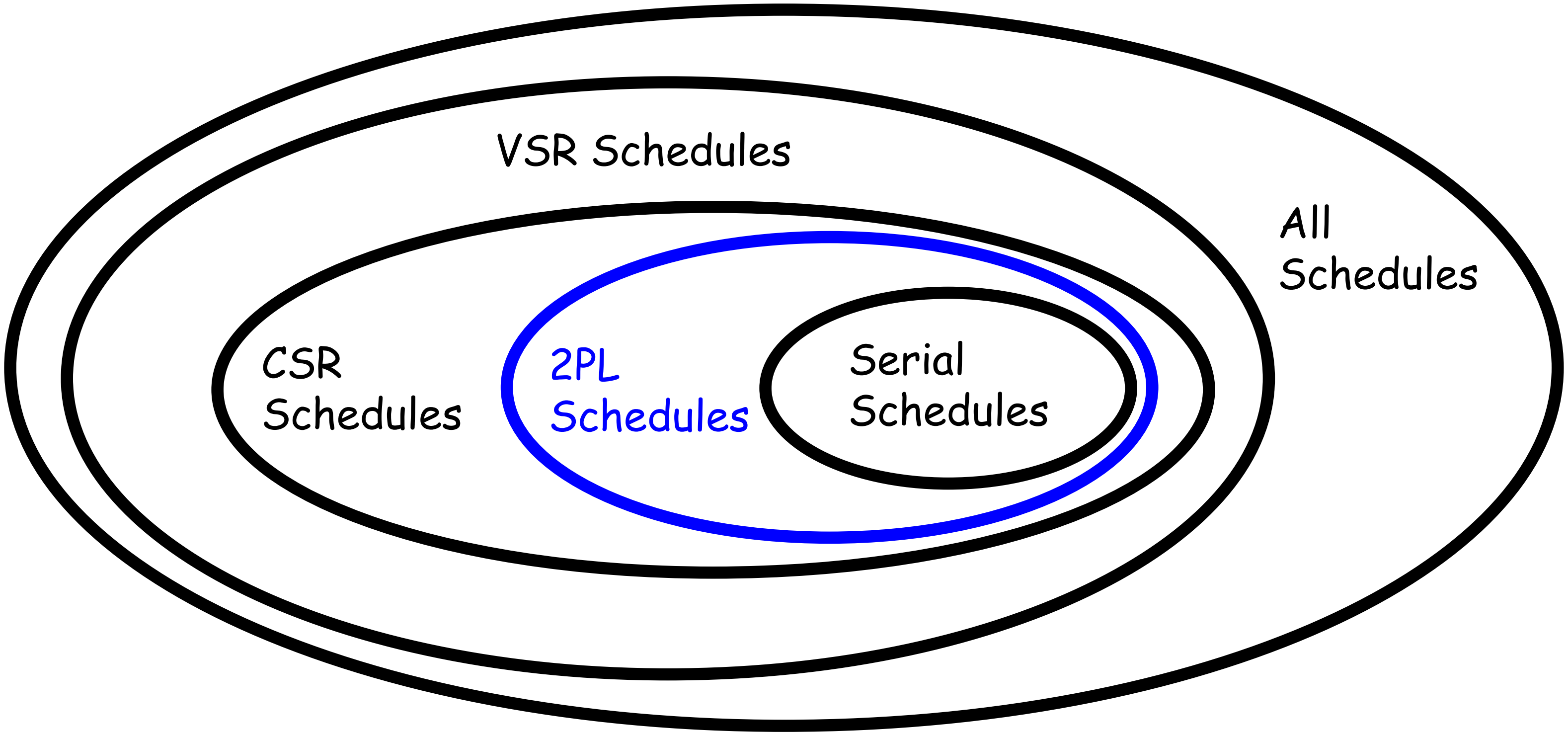
2PL implies CSR

- **2PL → CSR**: We assume that a schedule S is 2PL
- Consider, for each transaction, the end of the plateau
  - (i.e., the moment in which it holds all locks and is going to release the first one)
- We sort (and re-label) the transactions by this temporal value and consider the corresponding serial schedule S'
- In order to prove (by contradiction) that S' $\approx_C$ S ...

2PL implies CSR

- Consider a (generic) conflict $o_i \rightarrow o_j$ in S' with $o_i \in T_i$ $o_j \in T_j$ $i<j$
- By definition of conflict, o_i and o_j address the same resource **r**, and at least one of them is a write
- Can o_i and o_j occur in reverse order in S?
  - No, because then T_j should have released **r** *before* T_i could acquire it.
    - This contradicts the ordering criterion.
- Inclusion of 2PL in VSR descends from VSR $\supset$ CSR
- We proved that all **2PL schedules are view-serializable**
  - And they can be **checked with negligible overhead**

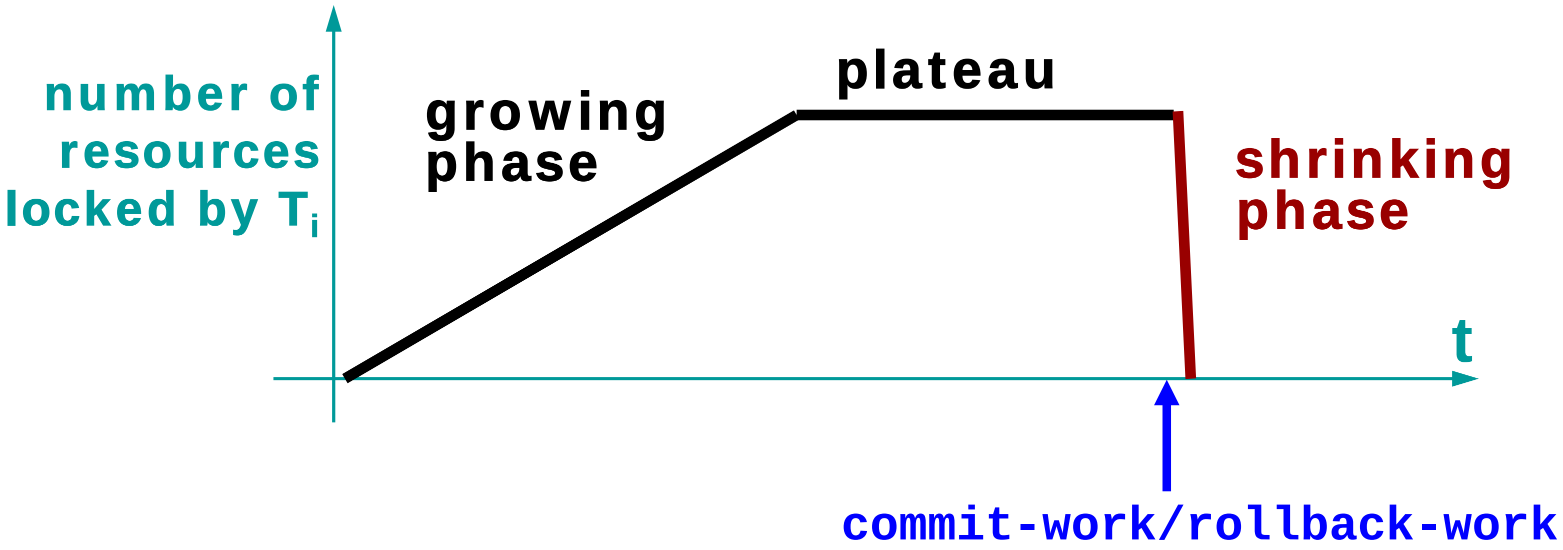
CSR, VSR and 2PL



Dirty reads are still a menace: Strict 2PL

- Up to now, we were still using the hypothesis of *commit-projection* (no transaction in the schedule aborts)
  - 2PL, as seen so far, does not protect against dirty reads (and therefore neither do VSR nor CSR)
    - Releasing locks before rollbacks exposes “dirty” data
- To remove this hypothesis, we need to add a constraint to 2PL, that defines **strict 2PL**:
  - *Locks held by a transaction can be released only* ***after*** *commit/rollback*
- This version of 2PL is used in commercial DBMSs

Strict 2PL in Practice



Isolation Levels in SQL:1999 (and JDBC)

- Writes are always applied **strict 2PL** (so update loss is avoided)
- Reads are possible with different guarantees:

- `READ UNCOMMITTED` allows dirty reads, nonrepeatable reads and phantoms:
  - No read lock (and ignores locks of other transactions)

- `READ COMMITTED` prevents dirty reads but allows nonrepeatable reads and phantoms:
  - Read locks (and complies with locks of other transactions), but without 2PL

Isolation Levels in SQL:1999 (and JDBC)

- **REPEATABLE READ** avoids dirty reads, nonrepeatable reads and phantom updates, but allows for phantom inserts:
  - 2PL also for reads, with data locks
- **SERIALIZABLE** avoids all anomalies:
  - 2PL with *predicate locks (on the relation or on the index)*

	Dirty Read	Non rep. read	Phantoms
Read uncomm.	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y (insert)
Serializable	N	N	N

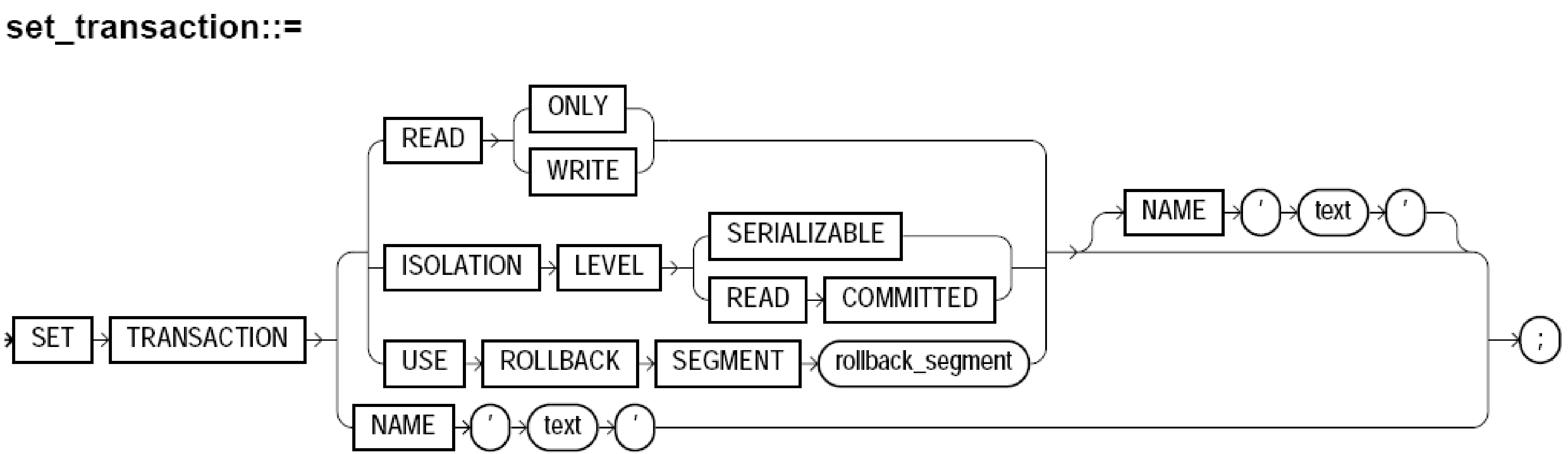
Predicate Locks

- A transaction T = **`update Tab set B=1 where A=1`**
- Then, the lock is on predicate A=1
  - Other transactions cannot insert, delete, or update any tuple satisfying this predicate
  - In the worst case (predicate locks not supported):
    - The lock extends to the entire table
  - In case the implementation supports predicate locks:
    - The lock is expressed with the help of indexes
- Predicate locks **protect from phantom inserts**, as the insertion of <u>new</u> data affecting a running query is forbidden

Tab

A	B

Transaction syntax in Oracle 10g



Read-only = only shared locks

Exercise

B. Concurrency control (5 p.)

Classify the following schedules with respect to VSR, CSR, 2PL, strict 2PL and TS multi: complete the table with YES/NO and motivate your answers.

	Schedule	VSR	CSR	2PL	Strict 2PL	TS Multi
1	r1(x), w1(x), r2(x), w2(x), r3(y), w3(y), r1(y), w1(y)					
2	r2(x), w2(x), r1(x), w1(x)					
3	r1(x), r2(x), w1(x), w2(x)					
4	r1(x), r2(x), w2(x), r2(y), r3(z) , w2(y), r1(y), w3(z)					

The impact of Locking: waiting is dangerous!

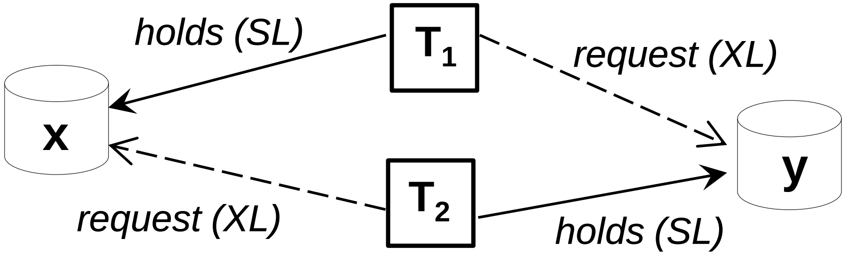
- Locks are tracked by ***lock tables*** (main memory data structures)
  - Resources can be *Free*, or *Read-Locked*, or *Write-locked*
  - To keep track of readers, every resource also has a "read counter"
- Transactions requesting locks are either **granted** the lock or **suspended and queued** (first-in first-out). There is risk of:
  - **Deadlock**: two or more transactions in endless (mutual) wait
    - Typically occurs when each transaction waits for another to release a lock (in particular: $r_1\ r_2\ w_1\ w_2$ → see *update lock* later)
  - **Starvation**: a single transaction in endless wait
    - Typically occurs due to write transactions waiting for resources that are continuously read (e.g., index roots)

Deadlock

- Occurs because concurrent transactions hold and, in turn, request resources held by other transactions

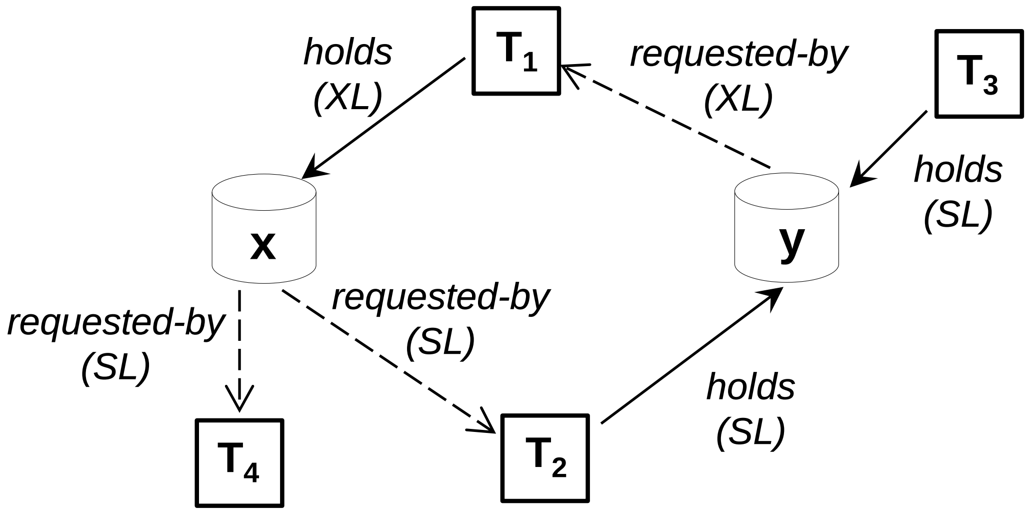
- T_1 : $r_1(x)$ $w_1(y)$
- T_2 : $r_2(y)$ $w_2(x)$

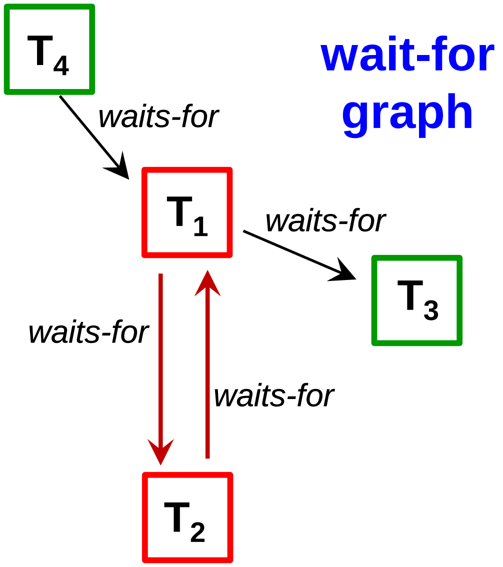
S: $r\_lock_1(x)$ $r\_lock_2(y)$ $r_1(x)$ $r_2(y)$ $w\_lock_1(y)$ $w\_lock_2(x)$



Deadlock

- A deadlock is represented by a cycle in the wait-for graph of transactions





Deadlock Resolution Techniques

- Timeout
  - Transactions killed after a long wait
    - How long?
- Deadlock prevention
  - Transactions killed when they COULD BE in deadlock
    - Heuristics
- Deadlock detection
  - Transactions killed when they ARE in deadlock
    - Inspection of the wait-for graph

Timeout Method

- A transaction is killed after a given amount of waiting (assumed as due to a deadlock)
  - The simplest method, widely used in the past
- The timeout value is system-determined (sometimes it can be altered by the database administrator)
- The problem is choosing a proper timeout value
  - Too long: useless waits whenever deadlocks occur
  - Too short: unrequired kills, redo overhead

Deadlock Prevention

Idea: killing transactions that ***could*** cause cycles

Resource-based prevention schemes: Restrictions on lock requests

- Transactions request all resources at once, and only once
- Resources are globally sorted and must be requested “in global order”
- <u>Problem</u>: it’s not easy for transactions to anticipate all requests!

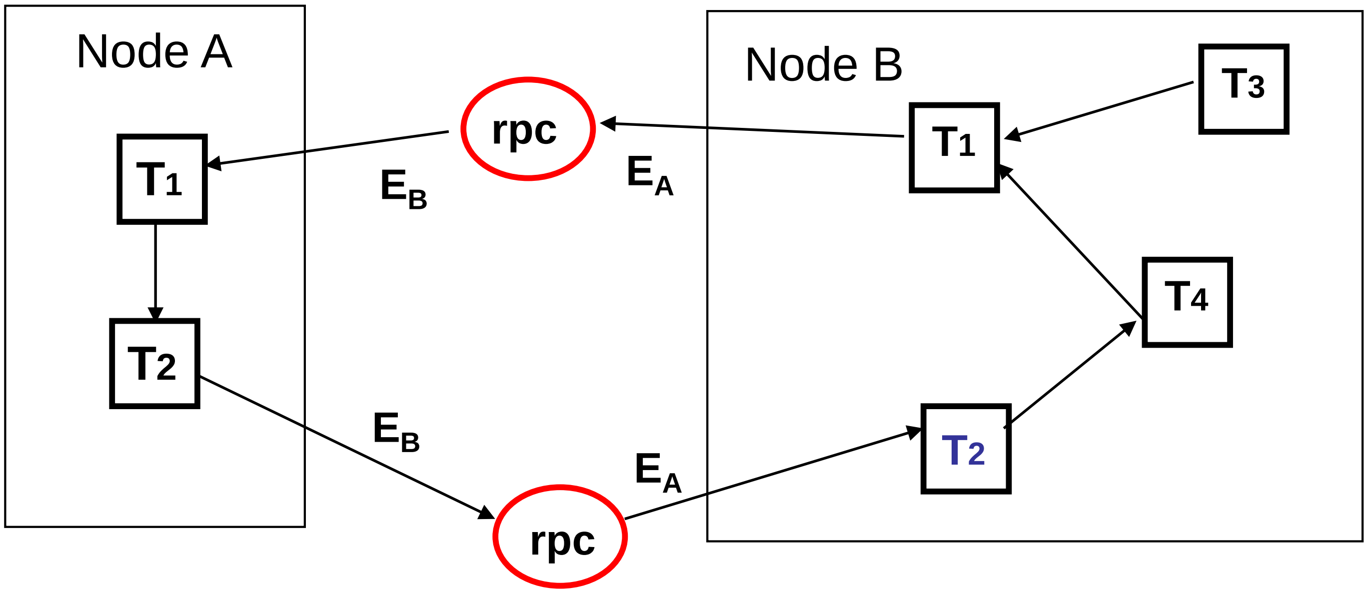
Transaction-based prevention schemes: Restrictions based on transactions’ IDs

- Assigning IDs to transactions (incrementally → transactions’ "age")
- Preventing "older" transactions from waiting for "younger" ones
- Options for choosing the transaction to kill
  - Pre-emptive (killing the waiting transaction – wound-wait)
  - Non-pre-emptive (killing the requesting transaction – wait-die)
- <u>Problem</u>: too many "killings"! (waiting probability >> deadlock probability)

Deadlock Detection

- Requires an algorithm to detect cycles in the wait-for graph
  - Must work with **distributed** resources
  - Must be efficient and reliable
- An elegant solution: **Obermark**'s algorithm (DB2-IBM, published on ACM-Transactions on Database Systems)
  - Assumes synchronous transactions, each executing on a single node and invoking “sub-transactions” on other nodes
  - Assumes communications via "remote procedure calls"
    - Both assumptions can be easily removed

Distributed Deadlock Detection: Problem Setting



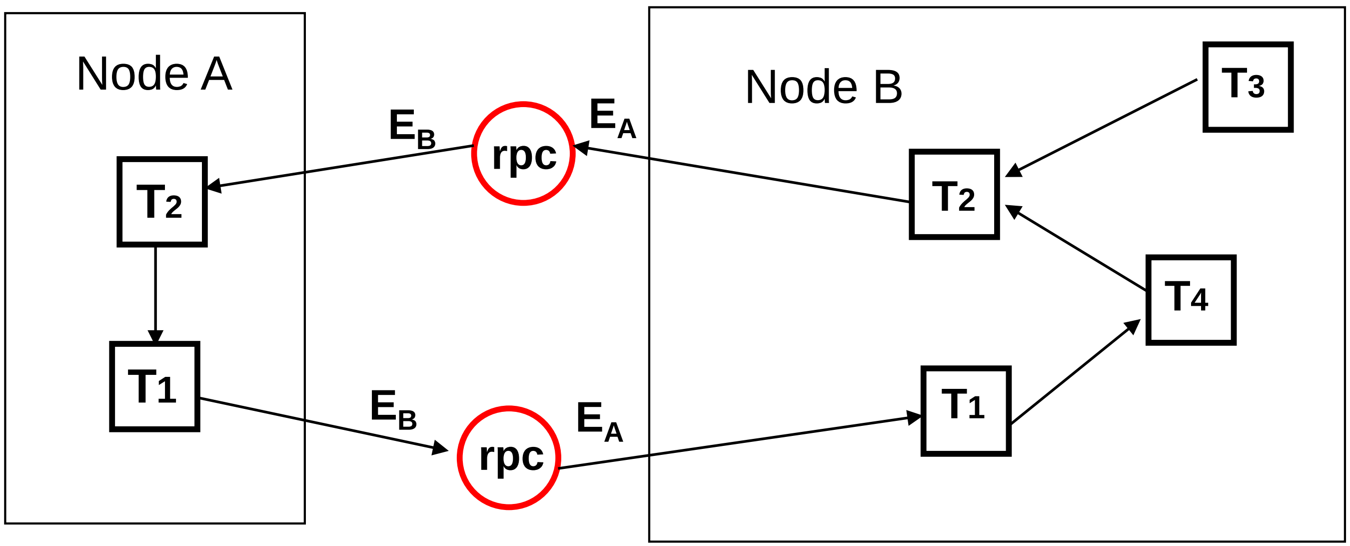
Potential Deadlock: at Node A: $E_B \rightarrow$ **$T_1 \rightarrow T_2$** $\rightarrow E_B$

at Node B: $E_A \rightarrow T_2 \rightarrow T_1 \rightarrow E_A$

Obermark's Algorithm

- Runs periodically at each node
- Consists of 4 steps
  - *Get potential deadlocks from the "previous" nodes*
  - *Integrate new arcs into the local wait-for graph*
  - *Search deadlocks: if found, kill one transaction*
  - *Send updated potential deadlocks to the "next" nodes*
- Secondary objective: detect each cycle only once; achieved by defining "previous" and "next" nodes, as follows:
  - Potential deadlocks transmitted only along RPC chains
  - Potential deadlock $E_x \rightarrow T\mathbf{i} \rightarrow T\mathbf{j} \rightarrow E_y$ is transmitted only if **i**>**j**

Algorithm execution, 1



Potential Deadlock at Node A: $E_B \rightarrow T_2 \rightarrow T_1 \rightarrow E_B$ sent to Node B

at Node B: $E_A \rightarrow T_1 \rightarrow T_2 \rightarrow E_A$ **not** sent (i<j)

Algorithm execution, 2

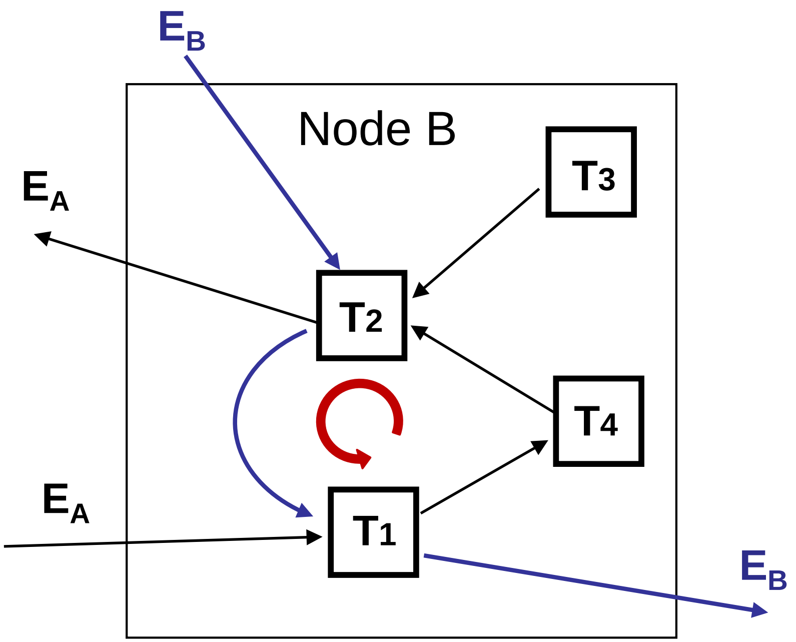
at Node B:

1. $E_B \rightarrow T_2 \rightarrow T_1 \rightarrow E_B$
   is received

2. $E_B \rightarrow T_2 \rightarrow T_1 \rightarrow E_B$ is added
   to the local wait-for graph

3. Deadlock detected:

   T_1 or T_2 or T_2 killed (rollback)



Another example

- Initially: at Node A, $E_C \rightarrow T_3 \rightarrow T_2 \rightarrow E_B$
  at Node B, $E_A \rightarrow T_2 \rightarrow T_1 \rightarrow E_C$
  at Node C, $E_B \rightarrow T_1 \rightarrow T_3 \rightarrow E_A$
- Nodes A and B can send messages (i>j), Node C cannot
- We assume that node A is the first to execute, sending to B
- At node B, a new potential dedadlock $E_C \rightarrow T_3 \rightarrow T_1 \rightarrow E_C$ is found by combining the incoming message $E_C \rightarrow T_3 \rightarrow T_2 \rightarrow E_B$ with the locally available conditions $T_2 \rightarrow T_1 \rightarrow E_C$. This is sent to node C.
- At node C, $E_C \rightarrow T_3 \rightarrow T_1 \rightarrow E_C$ is received and combined with local $T_1 \rightarrow T_3$. The dedalock is found and either T_1 or T_3 is killed.

Obermark: immateriality of conventions

There are two arbitrary choices in the algorithm:

- Send messages only if: (**1**) i > j vs. (**2**) i < j
- Send them to: (**a**) the following node vs. (**b**) the preceding node

- Therefore, there are four versions/variants of the algorithm
  (1+a), (1+b), (2+a), (2+b)
  - The sequence of the sent messages is different
  - However, they all identify deadlocks (if present)

Deadlocks in practice

- Their probability is much less than the conflict probability
  - Consider a file with n records and two transactions doing two accesses to their records (uniform distribution); then:
    - Conflict probability is O(**1 / n**)
    - Deadlock probability is O(**1 / n²**)
- Still, they do occur (once every minute in a mid-size bank)
  - The probability is linear in the *number* of transactions, quadratic in their *length* (measured by the number of lock requests)
    - Shorter transactions are healthier (ceteris paribus)
- There are techniques to limit the frequency of deadlocks
  - Update Lock, Hierarchical Lock, ...,

Update Lock

- The most frequent deadlock occurs when 2 concurrent transactions start by reading the same resource (SL) and then decide to write it and try to upgrade their lock to XL
- To avoid this situation, systems offer the UPDATE LOCK (UL) – asked by transactions that will read and then write an item

	Resource status		
Request	SL	UL	XL
SL	OK	**OK**	No
UL	**OK**	**No**	No
XL	No	No	No

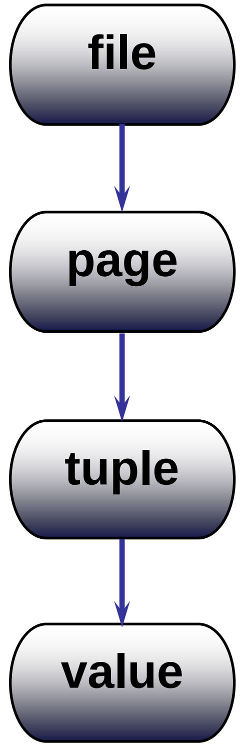
Hierarchical Locking

- In many real systems, locks are specified with different granularities
  - e.g.: schema, table, fragment, page, tuple, field
- These resources are in a hierarchy (or in a DAG)
- The choice of the lock granularity depends on the application:
  - Too coarse: too many locked resources, little concurrency
  - Too fine: too many lock requests, too much overhead

Resources Managed through Hierarchical Locking

lock at the
level (granularity) of:

file
page
tuple
attribute value



Reduced Granularity

Increased Concurrency

Hierarchical Locking

- Concept:
  - Locking can be done upon objects at various levels of granularity
- Objectives:
  - Setting the minimum number of lockings
  - Recognizing conflicts as soon as possible
- Organization: asking locking upon resouces organized as a hierarchy
  - Requesting resources top-down until the right level is obtained
  - Releasing locks bottom-up

Enhanced Locking Scheme

- 5 Lock modes:
  - In addition to read lock (**r-lock**) and write lock (**w-lock**), renamed for historical reasons into Shared Locks (**SL**) and Exclusive Locks (**XL**)
- The new modes express the "**intention** of locking at lower levels of granularity"
  - **ISL**: Intention of locking a subelement in shared mode
  - **IXL**: Intention of locking a subelement in exclusive mode
  - **SIXL**: Lock of the element in shared mode with intention of locking a subelement in exclusive mode (SL+IXL)

Conflicts in Hierarchical Locks

Request	Resource state ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Hierarchical Locking Protocol

- Locks are requested starting from the root and going down in the hierarchy
- Locks are released starting from the leaves and going up in the hierarchy
- To request an SL or ISL lock on a non-root element, a transaction must hold an ISL or IXL lock on its “parent”
- To request an IXL, XL or SIXL lock on a non-root element, a transaction must hold a SIXL or IXL lock on its “parent”

Example

P1	P2
t1	t5
t2	t6
t3	t7
t4	t8

P1	P2
t1	t5
t2	t6
t3	t7
t4	t8

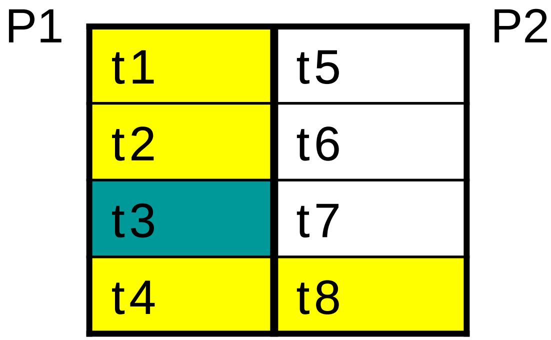
Page 1 (P1): t1,t2,t3,t4 tuples
Page 2 (P2): t5,t6,t7,t8 tuples

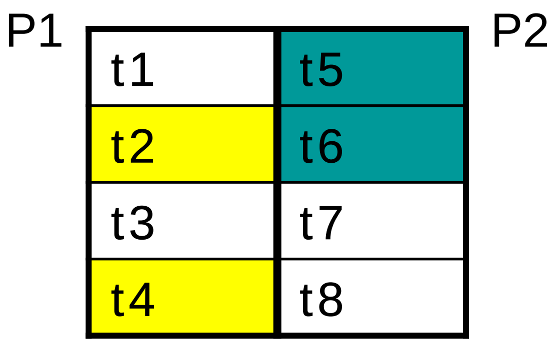
Transaction 1: read(P1)
write(t3)
read(t8)

Transaction 2: read(t2)
read(t4)
write(t5)
write(t6)

They are NOT in conflict!
(indipendently of the order)

Lock Sequences





Transaction 1:
IXL(root)
SIXL(P1)
XL(t3)
ISL(P2)
SL(t8)

Transaction 2:
IXL(root)
ISL(P1)
SL(t2)
SL(t4)
IXL(P2)
XL(t5)
XL(t6)

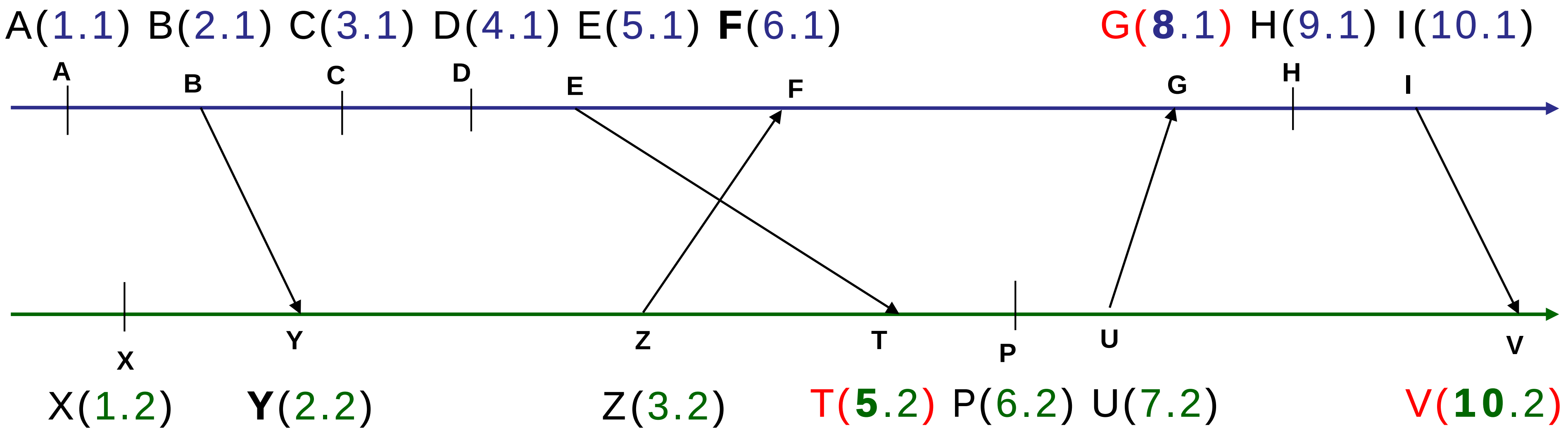
Concurrency Control Based on Timestamps

- Alternative to 2PL (and to locking in general)
- **Timestamp**:
  - *Identifier that defines a total ordering of the events of a system*
- Each transaction has a timestamp representing the time at which the transaction begins
- A schedule is accepted only if it reflects the serial ordering of the transactions induced by their timestamps

Assigning timestamps

- Timestamp: an indicator of the "current time"
- Assumption: no "global time" is available
- Mechanism: a system’s function gives out timestamps on requests
- Syntax: timestamp = event-id.node-id
  - event-ids are unique at each node
- Synchronization: send-receive of messages
  - for a given message m, send(m) precedes receive(m)
- Algorithm: cannot receive a message from "the future", if this happens the "bumping rule" is used to bump the timestamp of the *receive* primitive beyond the timestamp of the *send* primitive

Example of timestamp assignment



Events **Y** and **F** represent messages received "from the past" → OK

Events T, G and V represent messages received "from the future"

→ The local timestamp is incremented (*bumped*) accordingly

Timestamp-based concurrency control principles

- The scheduler has two counters: RTM(x) and WTM(x) for each object
- The scheduler receives read/write requests tagged with timestamps:
  - *read(x,ts)*:
    - If ts < WTM(x) the request is **rejected** and the transaction is killed
    - Else, access is **granted** and RTM(x) is set to max(RTM(x), ts)
  - *write(x,ts)*:
    - If ts < RTM(x) or ts < WTM(x) the request is **rejected** and the transaction is killed
    - Else, access is **granted** and WTM(x) is set to ts
- Many transactions are killed
- To work w/o the commit-projection hypothesis, it needs to "buffer" write operations until commit, which introduces waits

Example

Assume RTM(x) = 7

WTM(x) = 4

Request	Response	New value
read(x,6)	ok	
read(x,8)	ok	RTM(x) = 8
read(x,9)	ok	RTM(x) = 9
write(x,8)	no	T_8 killed
write(x,11)	ok	WTM(x) = 11
read(x,10)	no	T_{10} killed

2PL vs. TS

- They are incomparable
  - Schedule in TS but not in 2PL

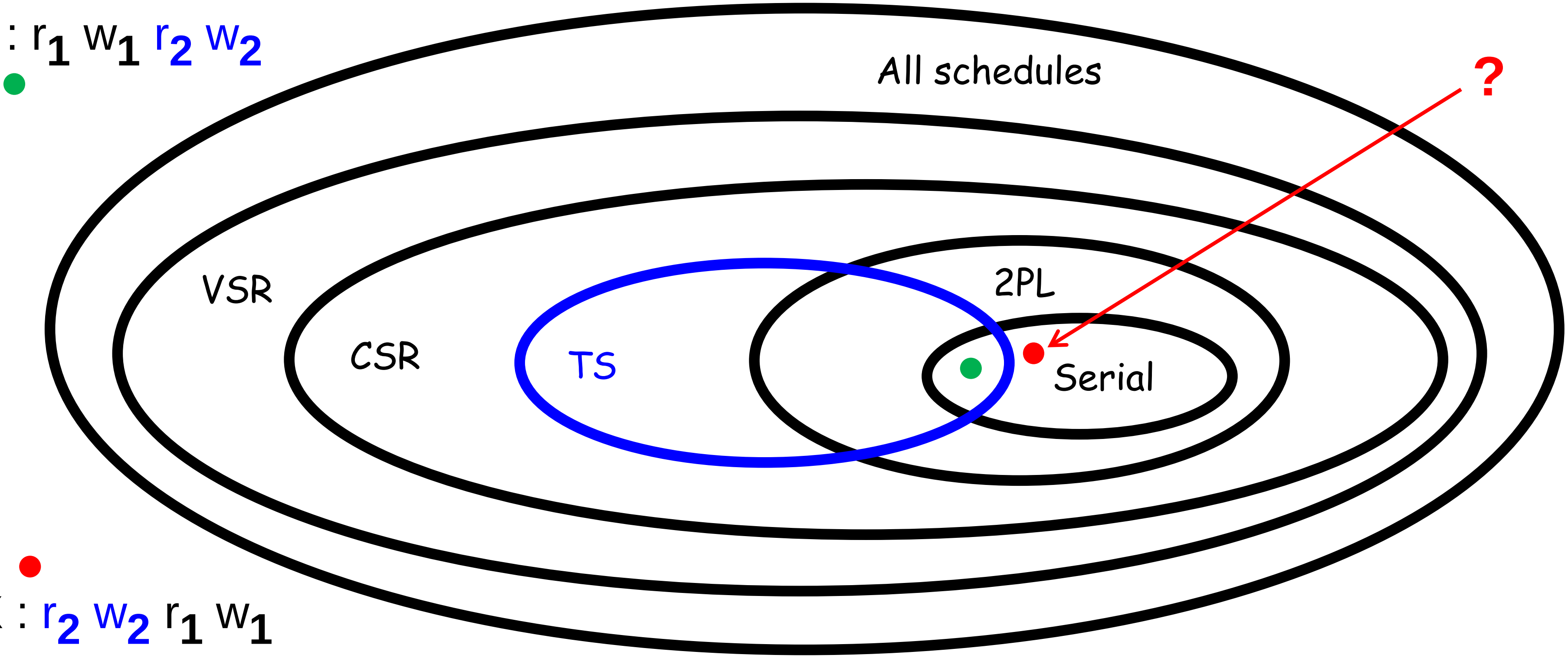
    $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_0(y)\ w_1(y)$
  - Schedule in 2PL but not in TS

    $r_2(x)\ w_2(x) r_1(x)\ w_1(x)$
  - Schedule in TS and in 2PL

    $r_1(x)\ r_2(y)\ w_2(y)\ w_1(x)\ r_2(x)\ w_2(x)$
- Besides: $r_2(x)\ w_2(x)\ r_1(x)\ w_1(x)$ is serial but not in TS

CSR, VSR, 2PL and TS

X : $r_1\ w_1\ r_2\ w_2$



X : $r_2\ w_2\ r_1\ w_1$

2PL vs. TS

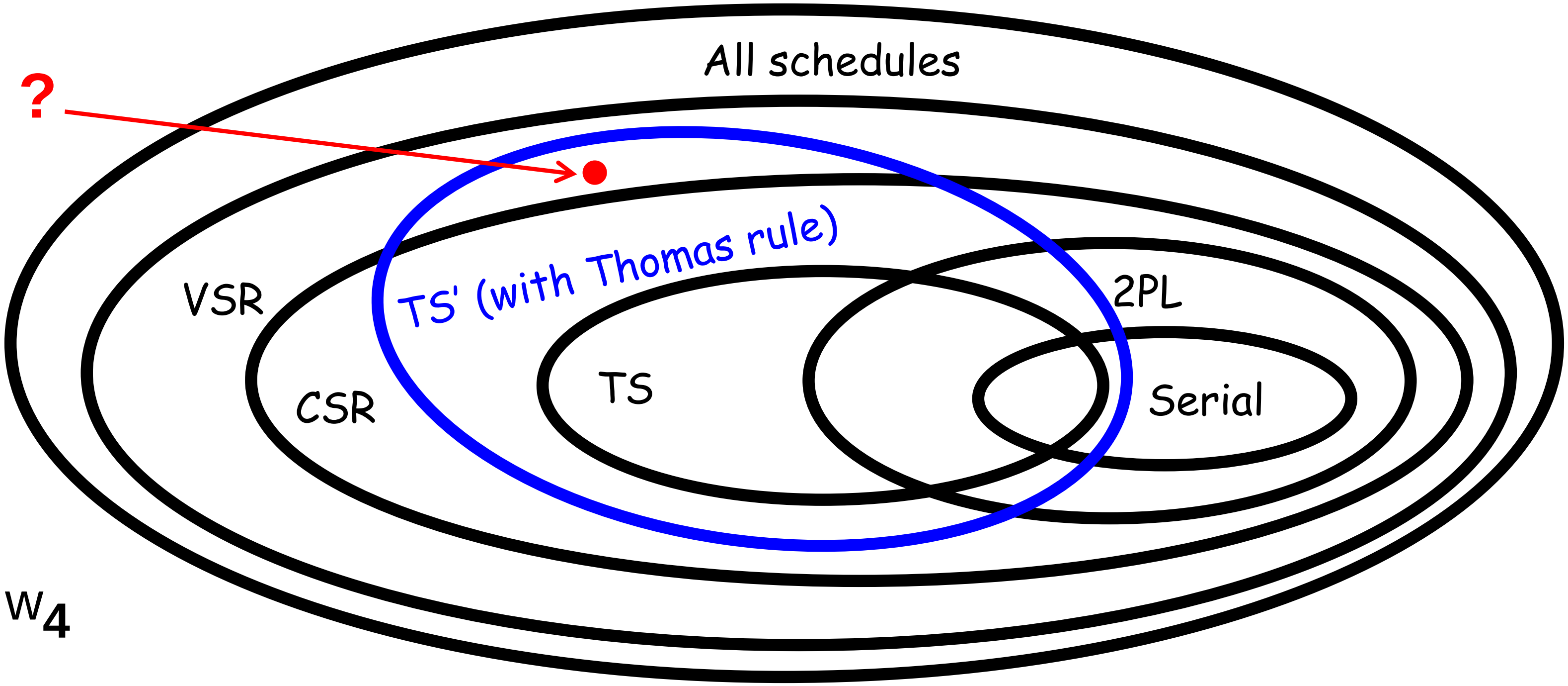
- In *2PL* transactions can be actively waiting. In *TS* they are killed and restarted
- The serialization order with *2PL* is imposed by conflicts, while in *TS* it is imposed by the timestamps
- The necessity of waiting for commit of transactions causes long delays in *strict 2PL*
- *2PL* can cause deadlocks
  - but also *TS,* if care is not taken
- Restarting a transaction costs more than waiting: *2PL* wins!

TS-based concurrency control: a variant (Thomas Rule)

- The scheduler has two counters: RTM(*x*) and WTM(*x*) for each object
- The scheduler receives read/write requests tagged with timestamps:
  - *read(x,ts)*:
    - If *ts* < WTM(*x*) the request is **rejected** and the transaction is killed
    - Else, access is **granted** and RTM(*x*) is set to max(RTM(*x*), *ts*)
  - *write(x,ts)*:
    - If *ts* < RTM(*x*) the request is **rejected** and the transaction is killed
    - Else, if *ts* < WTM(*x*) then our write is "obsolete": it can be **skipped**
    - Else, access is **granted** and WTM(*x*) is set to *ts*

- Does this modification affect the taxonomy of the serialization classes?

TS' (TS with Thomas Rule)



x: r_2 w_3

y: r_1 w_3 w_2 w_4

$r_1(y)$ $r_2(x)$ $w_3(y)$ $w_2(y)$ $w_3(x)$ $w_4(y)$

Multiversion Concurrency Control

- Idea: writes generate new copies, reads access the "right" copy
- Writes generate new copies, each one with a new WTM. Each object x always has $N>1$ active copies with $WTM_N(x)$. There is a unique global RTM(x)
- Old copies are discarded when there are no transactions that need these values

Multiversion Concurrency Control

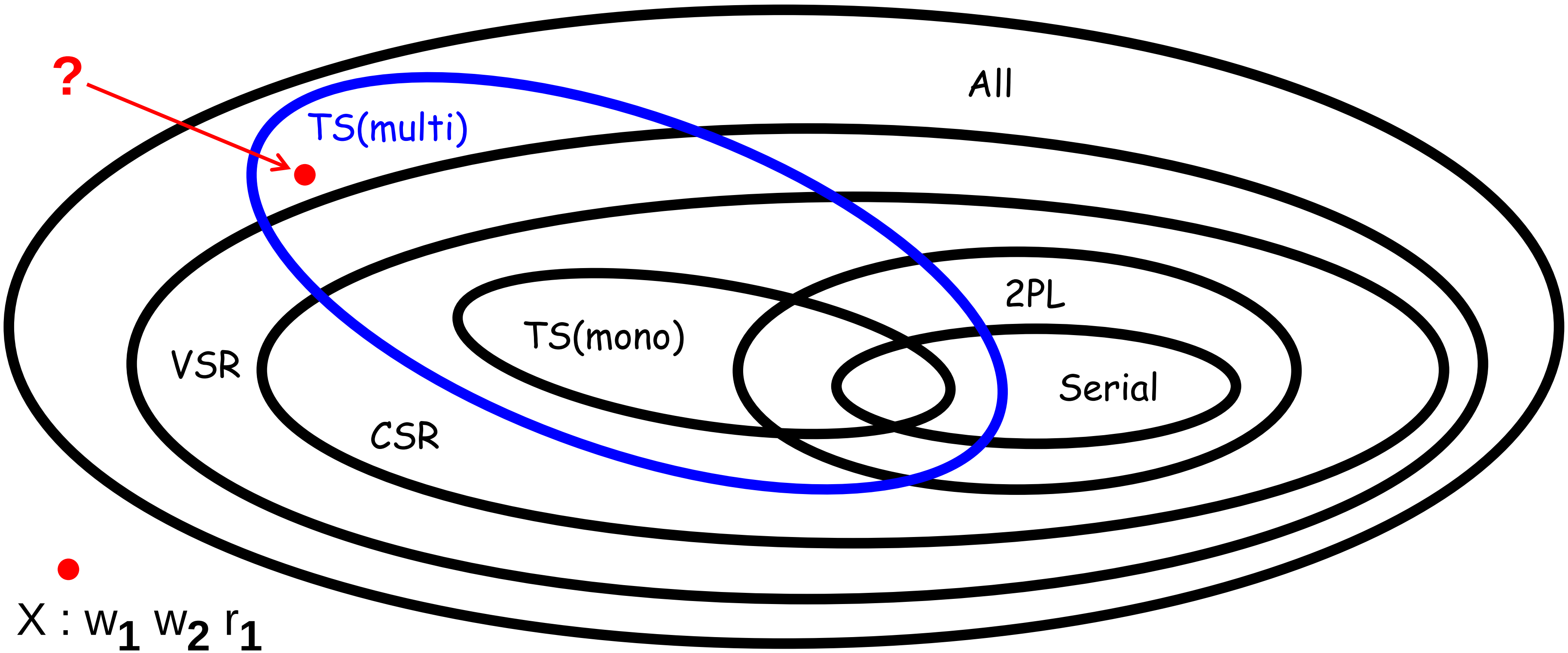
- Mechanism:
  - *read*(x,ts) is always accepted. A copy x_k is selected for reading such that:
    - If ts > $WTM_N(x)$, then k = N
    - Else take k such that $WTM_k(x) < ts < WTM_{k+1}(x)$
  - *write*(x,ts):
    - If ts < $RTM(x)$ the request is rejected
    - Else a new version is created (N is incremented) with $WTM_N(x) = ts$

Example

Assume RTM(x) = 7 N=1 WTM(x_1) = 4

Request	Response	New Value
read(*x*,6)	ok	
read(*x*,8)	ok	RTM(*x*) = 8
read(*x*,9)	ok	RTM(*x*) = 9
write(*x*,8)	no	T_8 killed
write(*x*,11)	ok	N=2, WTM(x_2) = 11
read(x,10)	ok on x_1	***RTM(x) = 10***
read(*x*,12)	ok on x_2	*RTM(x) = 12*
write(*x*,13)	ok	N=3, WTM(x_3) = 13

CSR, VSR, 2PL, TSmono, TSmulti



X : w_1 w_2 r_1

Snapshot Isolation (SI)

- The realization of multi-TS gives the opportunity to introduce into SQL another isolation level, **SNAPSHOT ISOLATION**
- In this level, no RTM is used on the objects, only WTMs
- Every transaction reads the version consistent with its timestamp (snapshot), and defers writes to the end
- If a transaction notices that its writes damage writes occurred after the snapshot, it aborts
  - It is called an optimistic approach

Anomalies in Snapshot Isolation

- Snapshot isolation does not guarantee serializability

T_1: update Balls set Color=White where Color=Black

T_2: update Balls set Color=Black where Color=White

- Serializable executions of T_1 and T_2 will produce a final configuration with balls that are either all white or all black
- An execution under Snapshot Isolation in which the two transactions start with the same «snapshot» will just swap the two colors