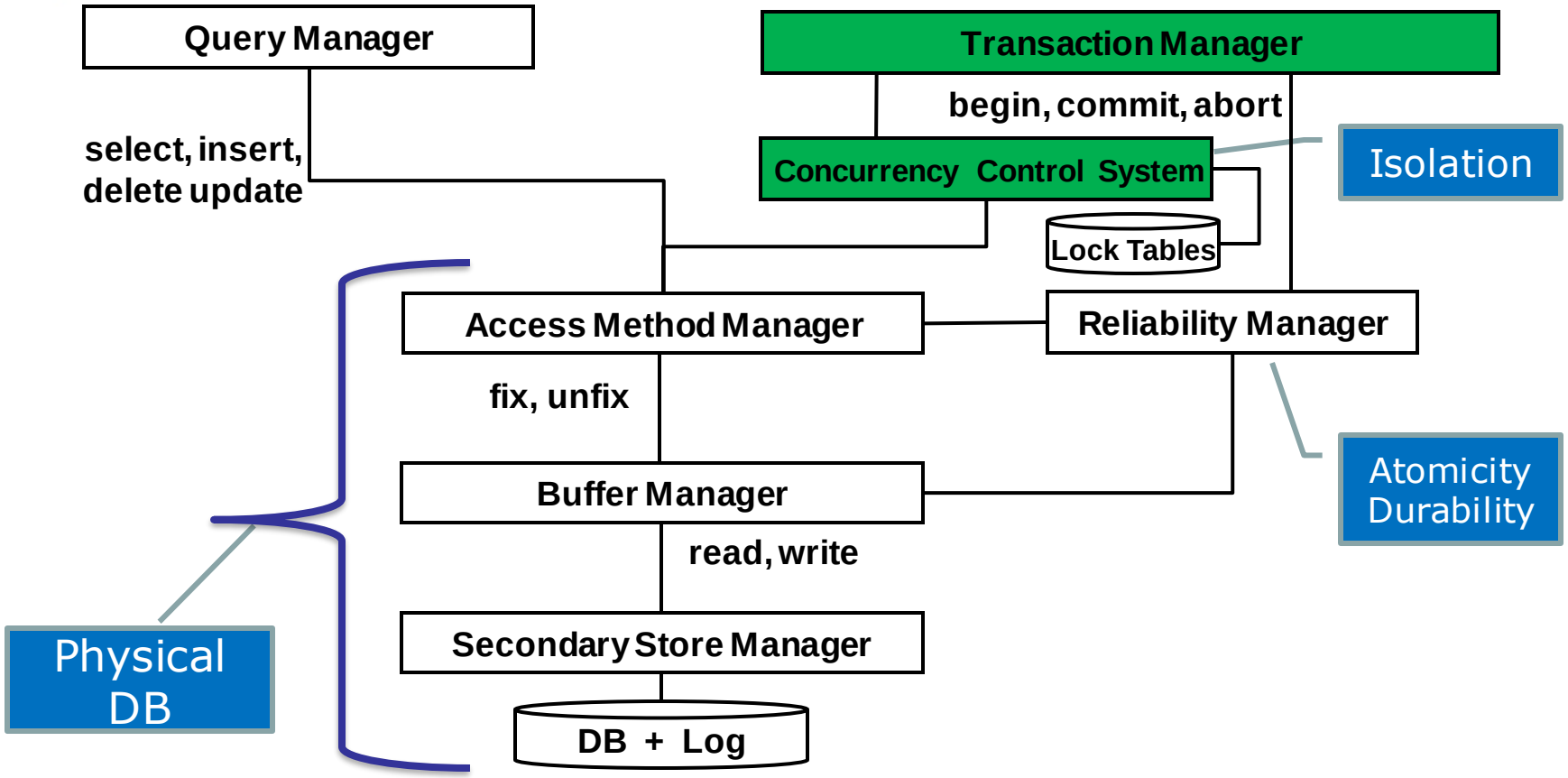
Databases 2

2 Concurrency Control

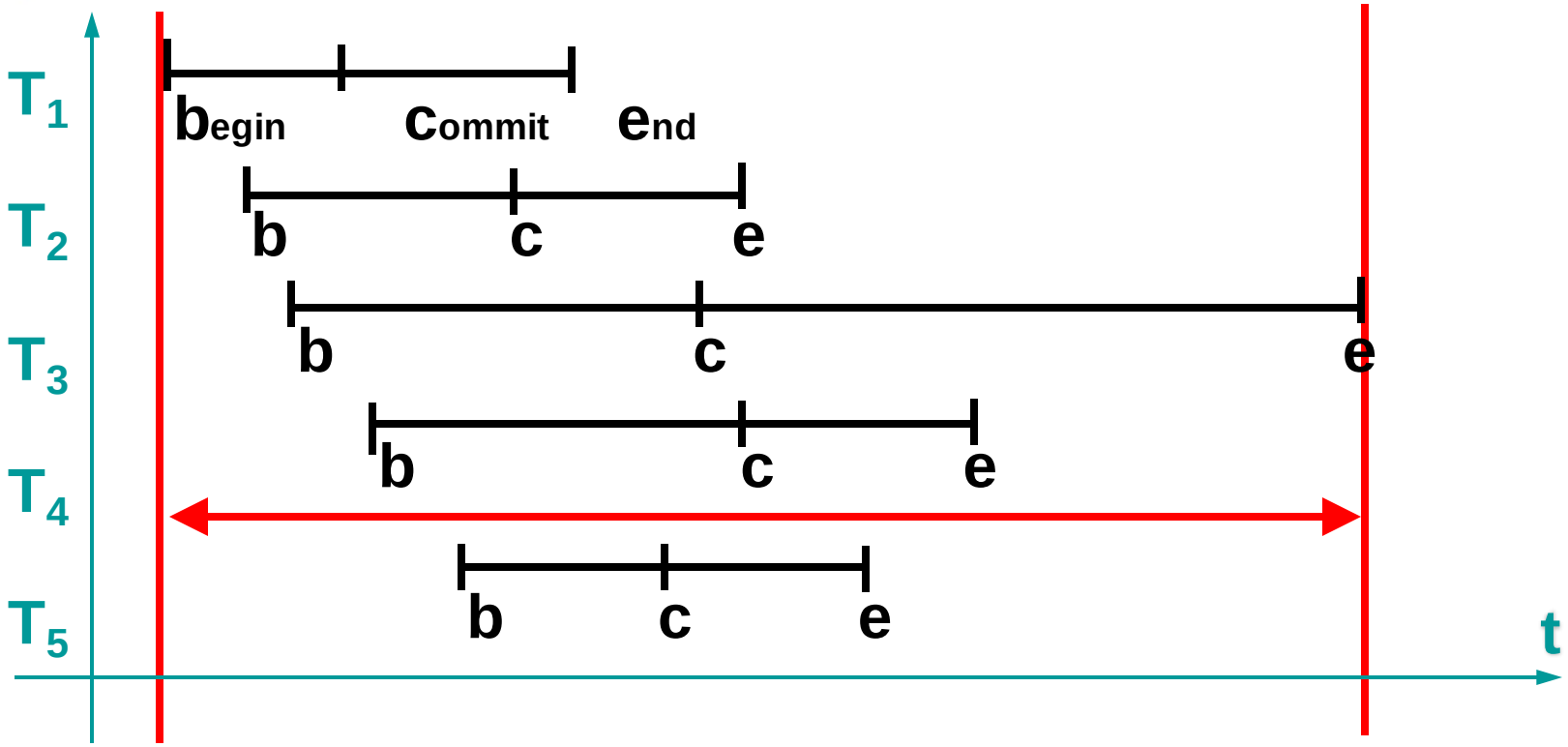
Concurrency Control

Concurrency In the DBMS architecture



- The Concurrency Control System
  - Manage the simultaneous execution of transactions
  - Avoids the insurgence of anomalies
  - While ensuring performance

Advantages of Concurrency



- Goal: exploit parallelism to maximise transactions per second (TPS)

Problems due to Concurrency

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

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

Concurrent SQL transactional statements addressing the same resource

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

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

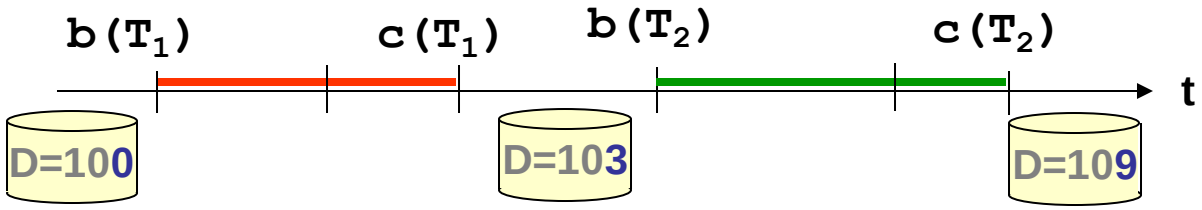
T_1 :
```
begin transaction
read(D → x)
x = x + 3
write(x → D)
commit work
```

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

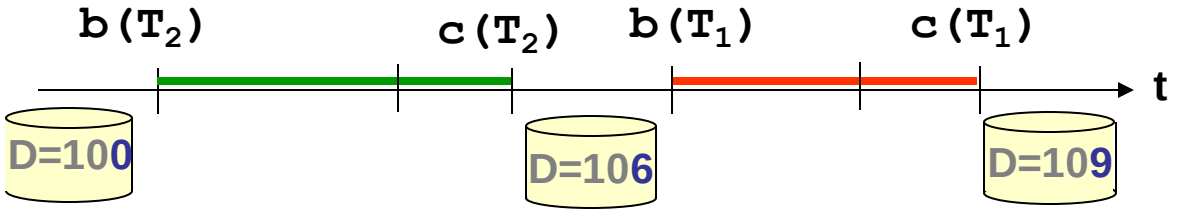
Serial Executions

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

Initially D_0=100




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

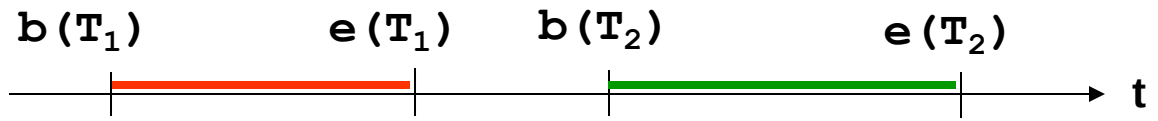


Concurrency Control

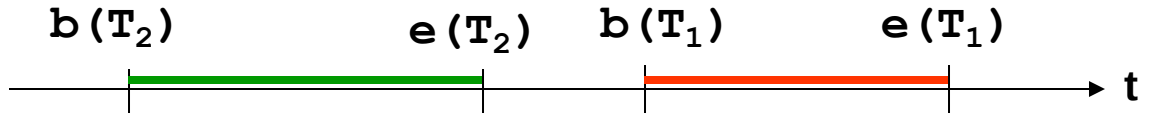
- Concurrency is fundamental
  - Tens or hundreds of transactions per second cannot be executed serially
- Examples: banks, ticket reservations

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

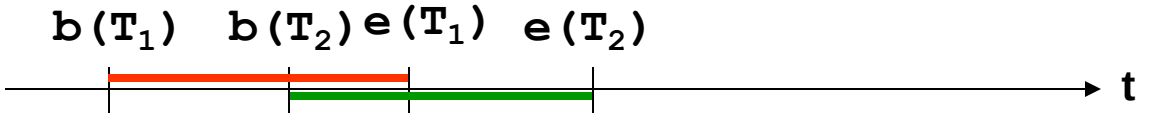
Concurrent Executions



SERIAL



SERIAL



INTERLEAVED

NESTED

Concurrency Control

Execution with Lost Update

D = 100

1	T_1: r(D→ x)	
2	T_1: x = x + 3	
3	T_2: r(D → y)	
4	T_2: y = y + 6	
5	T_1: w(x → D)	D = 103
6	T_2: w(y → D)	D = 106 !

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

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

- Note: this anomaly does not depend merely on T2 overwriting the value produced by T1
  - w1(x), w2(x) is ok (serial)
  - r1(x), w1(x), w2(x) is ok too (serial)
  - r1(x), r2(x), w1(x), w2(x) is not ok: inconsistent updates from the same initial value

Sequence of I/O Actions producing the Error

or

Concurrency Control

"Dirty" Read

D = 100

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

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

1 T_1: r(D → x)
2 T_1: x = x + 3
3 T_1: w(x → D) D = 103
4 T_2: r(D → y) this read is "dirty" (uncommitted value)
5 T_1: **rollback**
6 T_2: y = y + 6
7 T_2: w(y → D) D = 109 **!**

Concurrency Control

"Nonrepeatable" Read

D = 100

1 T_1: r(D → x)
2 T_2: r(D → y)
3 T_2: y = y + 6
4 T_2: w(y → D) D = 106
5 T_1: r(D → **z**) z <> x **!**

T1 : **SELECT balance FROM account**
WHERE client = 'Smith'
...
SELECT balance FROM account
WHERE client = 'Smith'

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

Phantom Update

A+B+C=100, A=50, B=30, C=20

T_1: r(A → x), r(B → y)……

T_2: r(B → s), r(C → t)

T_2: s = s + 10, t = t – 10

T_2: w(s → B), w(t → C) (now B=40, C=10, A+B+C=100)

T_1: r(C → z) (but, for T_1, x+y+z = A+B+C = **90!**)

So for T_1 it is as if "somebody else" had updated the value of the sum

But for T_2 the update is perfectly legal (does not change the value of the sum)

Phantom Insert

T_1: C=AVG(B: A=1) *C = select avg(B) from Tab where A=1*

T_2: Insert (A=1,B=2) *insert into Tab values(1, 2)*

T_1: C=AVG(B: A=1) *C = …*

Tab

A	B

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

Summary of Anomalies

- Lost update r_1 - r_2 - w_2 - w_1
  - An update is applied from a state that ignores a preceding update, which is lost
- Dirty read r_1 - w_1 - r_2 - $abort_1$ - w_2
  - An uncommitted value is used to update the data
- Nonrepeatable read r_1 - r_2 - w_2 - r_1
  - Someone else updates a previously read value
- Phantom update r_1 - r_2 - w_2 - r_1
  - Someone else updates data that contributes to a previously read datum
- Phantom insert r_1 - w_2(*new data*) - r_1
  - Someone else inserts data that contributes to a previously read datum

Concurrency Control

Concurrency Theory vs System Implementation

- **Model**: an **abstraction** of a system, object or process, which purposely **disregards details** to simplify the investigation of **relevant properties**
- Concurrency Theory builds upon a **model of transactions** and concurrency control principles that helps understanding real systems
- Real systems exploit implementation level mechanisms (locks, snapshots) which help achieve some of the desirable properties postulated by the theory
  - Don’t look for a view or conflict serializability checker in your DBMS
  - Look instead for lock tables, lock types, lock granting rules, snapshots, etc..
  - Understand how the implementation mechanisms ensure properties modelled by the Concurrency Theory

Transactions, Operations, Schedules

- Operation: *a read or write of a specific datum by a specific transaction*
  - NB: the schedule notation omits the program variables
    - $r_1(x) = r_i(x \rightarrow .)$
    - $w_i(x) = w_i(. \rightarrow x)$
    - $r_1(x)$ and $r_1(y)$ are different operations
    - $r_1(x)$ and $r_2(x)$ are different operations
    - $w_1(x)$ and $w_2(x)$ are different operations
- Schedule: *a sequence of operations performed by concurrent transactions that respects the order of operations of each transaction*

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:

$r_1(x)\ w_1(x)\ r_2(z)\ w_2(z)$	***serial***
$r_2(z)\ w_2(z)\ r_1(x)\ w_1(x)$	
$r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$	***interleaved***
$r_2(z)\ r_1(x)\ w_2(z)\ w_1(x)$	
$r_1(x)\ r_2(z)\ w_2(z)\ w_1(x)$	***nested***
$r_2(z)\ r_1(x)\ w_1(x)\ w_2(z)$	

$$N = 6$$

Schedules: how many?

- 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 … $o_1^{k_1}$
T_2: o_2^1 o_2^2 … $o_2^{k_2}$
…
T_i: o_i^1 o_i^2 … $o_i^{k_i}$
…
T_n: o_n^1 o_n^2 … $o_n^{k_n}$

$$N_S = n!$$

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

$$N_S << N_D$$

- N_s = number of permutations of N transactions
- N_D = number of permutations of all operations / product of the number of permutations of the operations of each transaction (only 1 out of the K_i permutations is valid, the one that respects the sequence of operations)

Principles of Concurrency Control

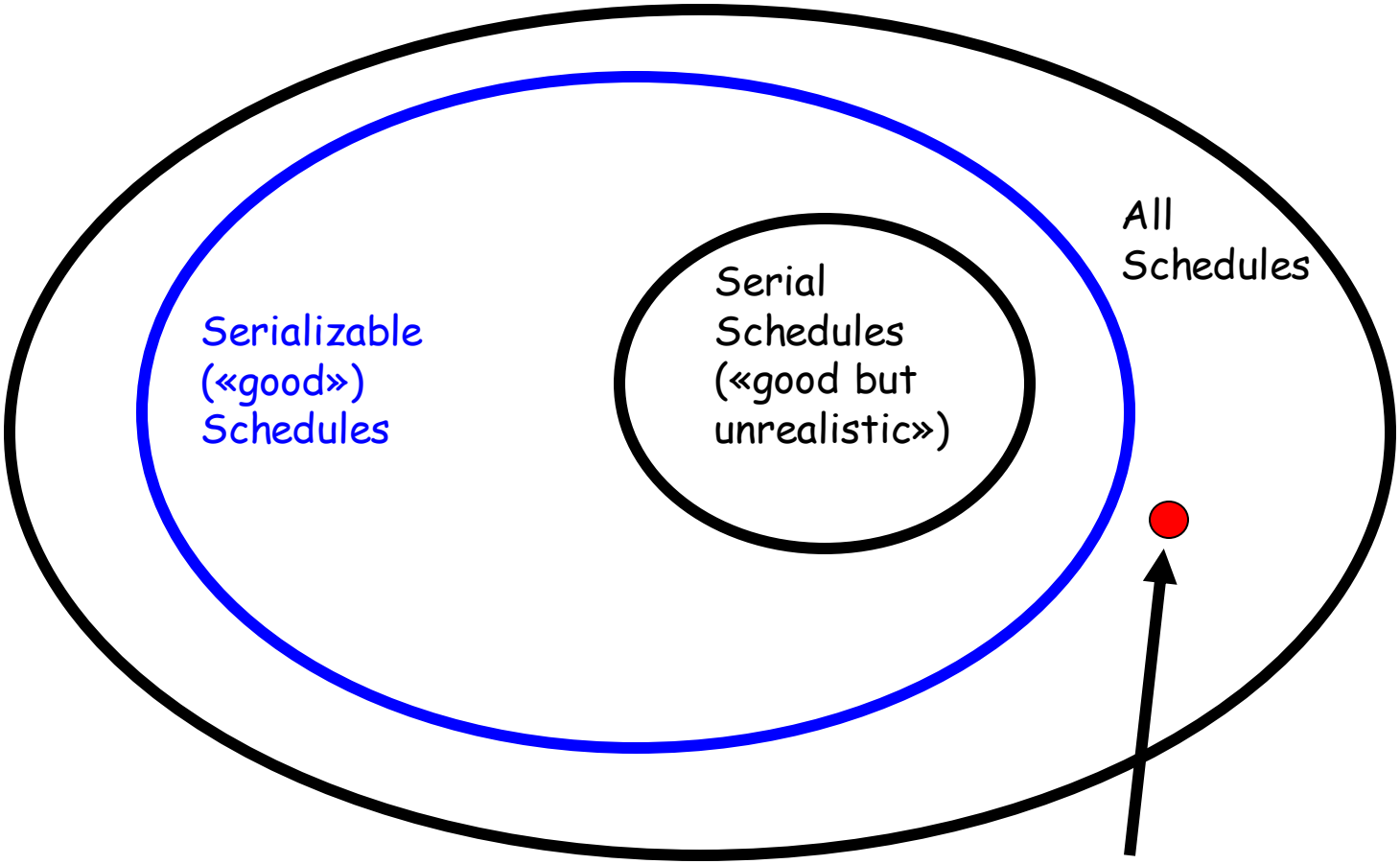
- **Goal**: to reject schedules that cause anomalies
- ***Scheduler***: a component that accepts or rejects the operations requested by the transactions
- ***Serial schedule***: a schedule in which the actions of each transaction occur in a contiguous sequence

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 leaves the database in the ***same state*** as **some** serial schedule of the same transactions
  - Commonly accepted as a notion of schedule **correctness**
  - Requires a notion of schedule equivalence to identify classes of schedules that ensure serializability
    - Different notions → different classes (cost of checking is schedule is serializable)
- Assumption
  - We initially assume that transactions are observed "a posteriori" limited to those that have committed (***commit-projection***), and we decide whether the observed schedule is admissible
  - In practice (and in contrast), schedulers must take decisions ***while transactions are running***

Basic Idea



This schedule **may** generate anomalies.
Its execution leads to a database state that no serial execution would produce

View-serializability

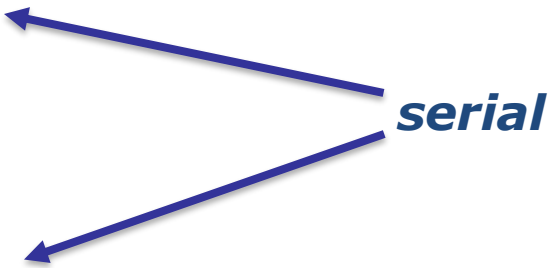
- Preliminary definitions:
  - $r_i(x)$ ***reads-from*** $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ and there is no $w_k(x)$ in S between $r_i(x)$ and $w_j(x)$
  - $w_i(x)$ in a schedule S is a ***final write*** if it is the last write on x that occurs in S
- Two schedules are ***view-equivalent*** ($S_i \approx_{\mathbf{v}} S_j$) if: they have the same operations, the same reads-from relation, and the same final writes
- 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**
- Mnemonically: S is ***view-serializable*** if 1) every read operation sees the same values and 2) the final value of each object is written by the same transaction as if the transactions were executed serially in **some** order

Examples of View-serializability

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

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

(switch of read operations on the same object)



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

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

(switch of read/write ops on different objects that preserves final write)

- S_3 is view-equivalent to serial schedule S_4 (so it is view-serializable)
- S_5 is not view-equivalent to S_4 (different operations) but 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

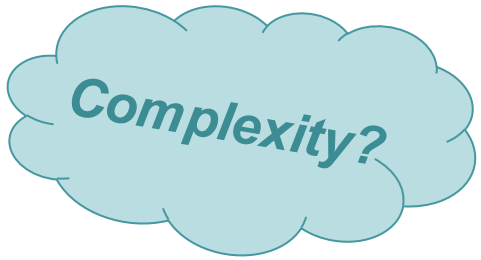
S_{10}: $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $w_0(y)$ $r_3(z)$ $w_3(z)$ $w_2(y)$ $w_1(x)$ $w_3(y)$

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: $w_1(x)$, $w_3(y)$, $w_3(z)$

$S_{10} \in VSR$



A More Complex Example

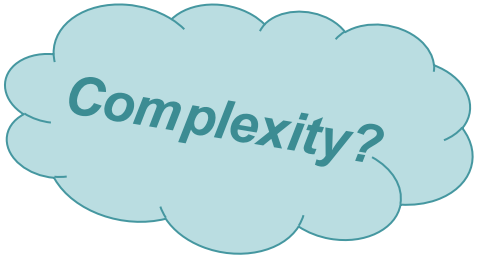
S_{10}: $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $w_0(y)$ $r_3(z)$ $w_3(z)$ $w_2(y)$ $w_1(x)$ $w_3(y)$

S_{13}: $w_0(x)$ $w_0(z)$ $w_0(y)$ $r_2(x)$ $w_2(y)$ $r_3(z)$ $w_3(z)$ $w_3(y)$ $r_1(x)$ $r_1(z)$ $w_1(x)$

reads-from OK: $r_1(x)$ from $w_0(x)$, $r_1(z)$ from $w_3(z)$

Different read from relation

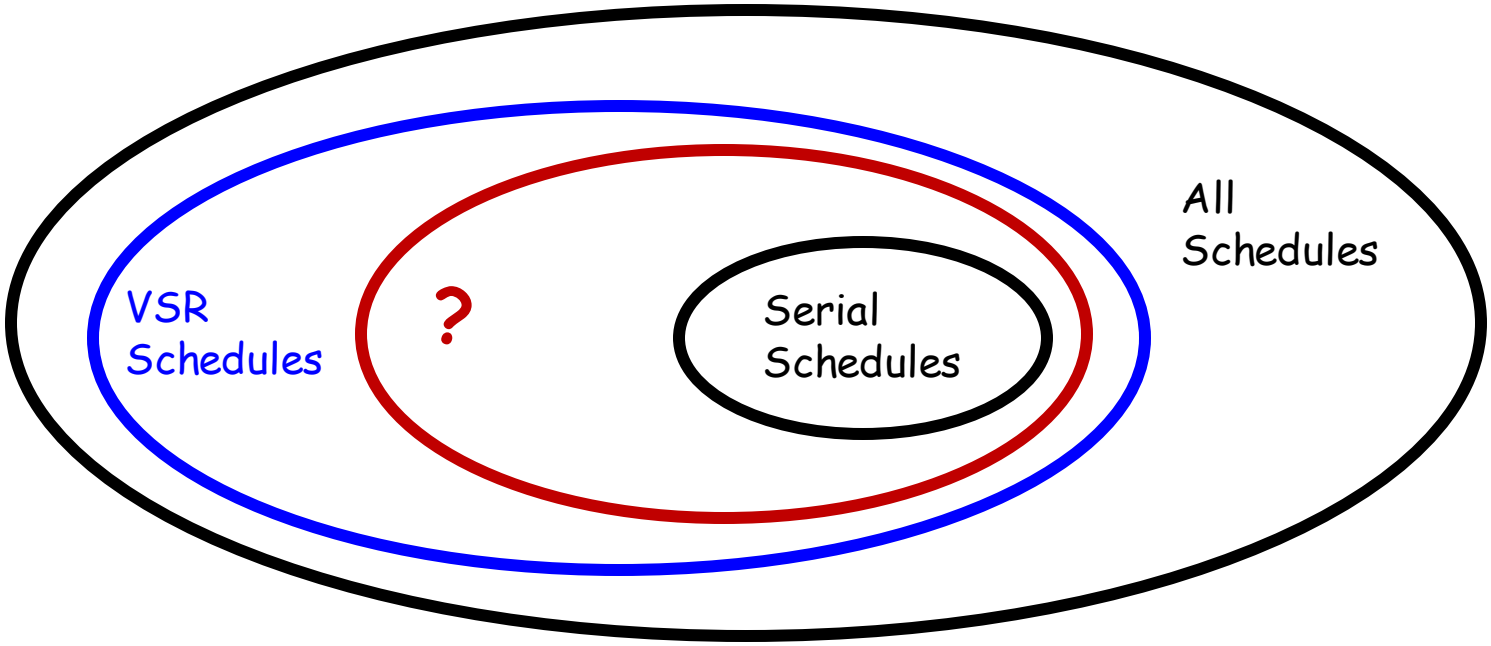
S10 not VSR-equivalent to T0,T2,T3, T1



Complexity of View-serializability

- Deciding view-equivalence of **two given** schedules is done in **polynomial** time and space
- Deciding if a **generic** schedule is in VSR 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!*
- We look for a stricter definition that is easier to check
  - May lead to rejecting some schedule that would be acceptable under view-serializability but not under the stricter-faster criterion

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 in
  all conflicting pairs transactions 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**

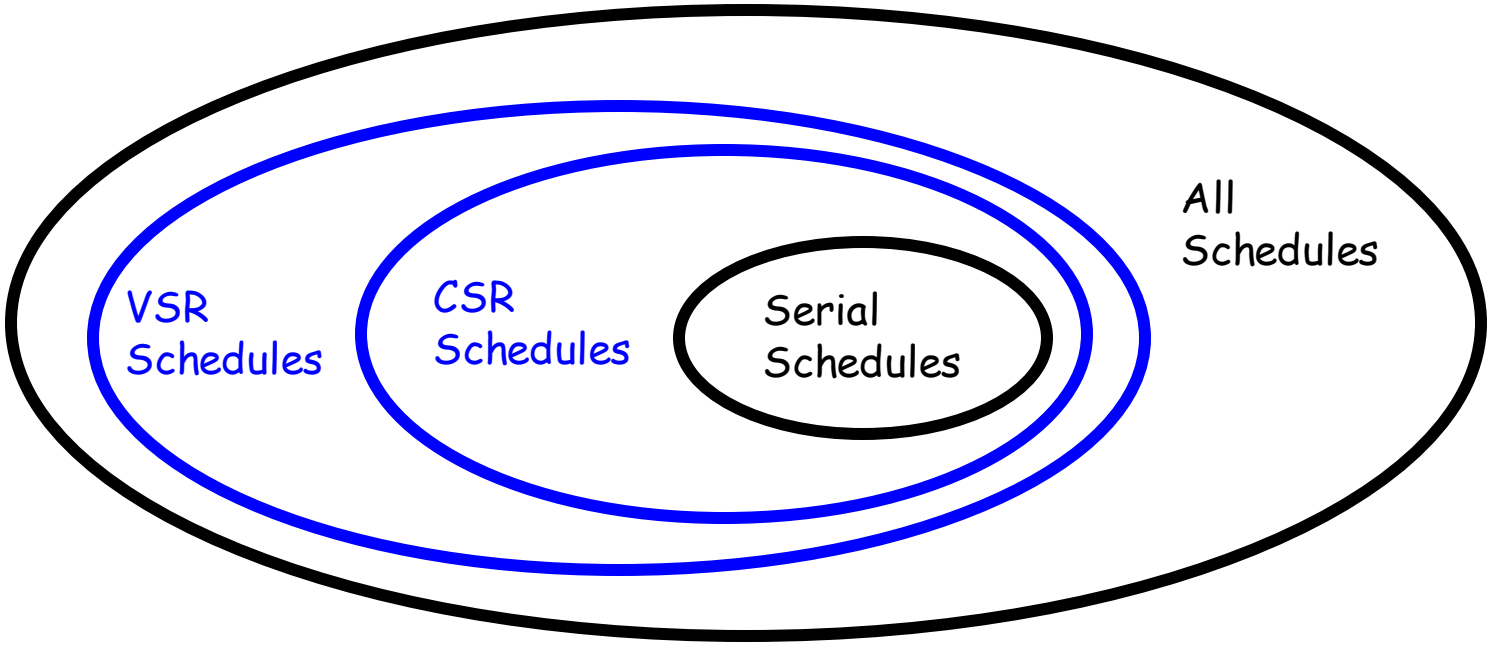
Relationship between CSR and VSR

- **VSR ⊃ 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 $T_1T_2T_3$ w.r.t. which it is view equivalent
    - nor $T_2T_1T_3$ which preserves the final write on x
    - nor all the others..

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

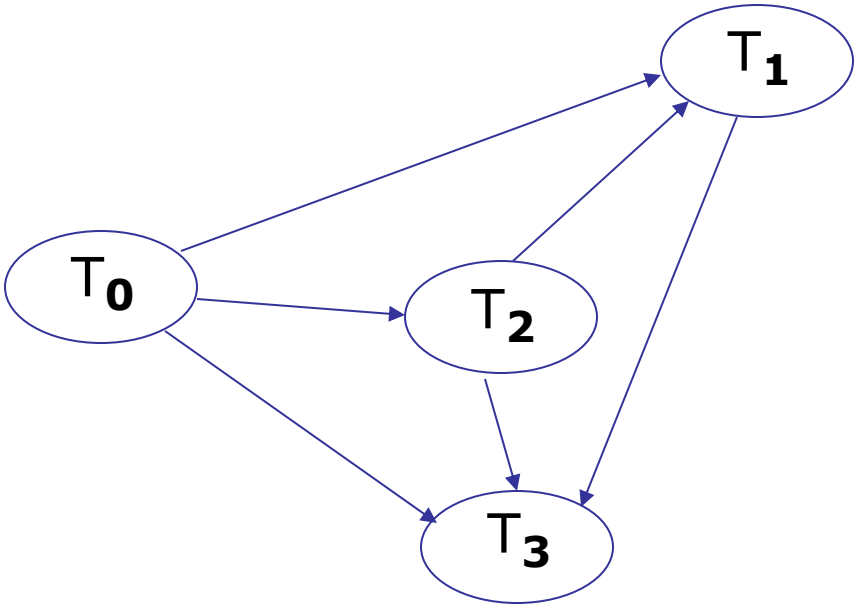
- Is done with a *conflict graph* that has:
  - One node for each transaction T_i
  - One arc from T_i to T_j if there exists at least one conflict between an operation o_i of T_i and an operation o_j of T_j such that o_i precedes o_j
- Theorem:
  - A schedule is in CSR if and only if its conflict graph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

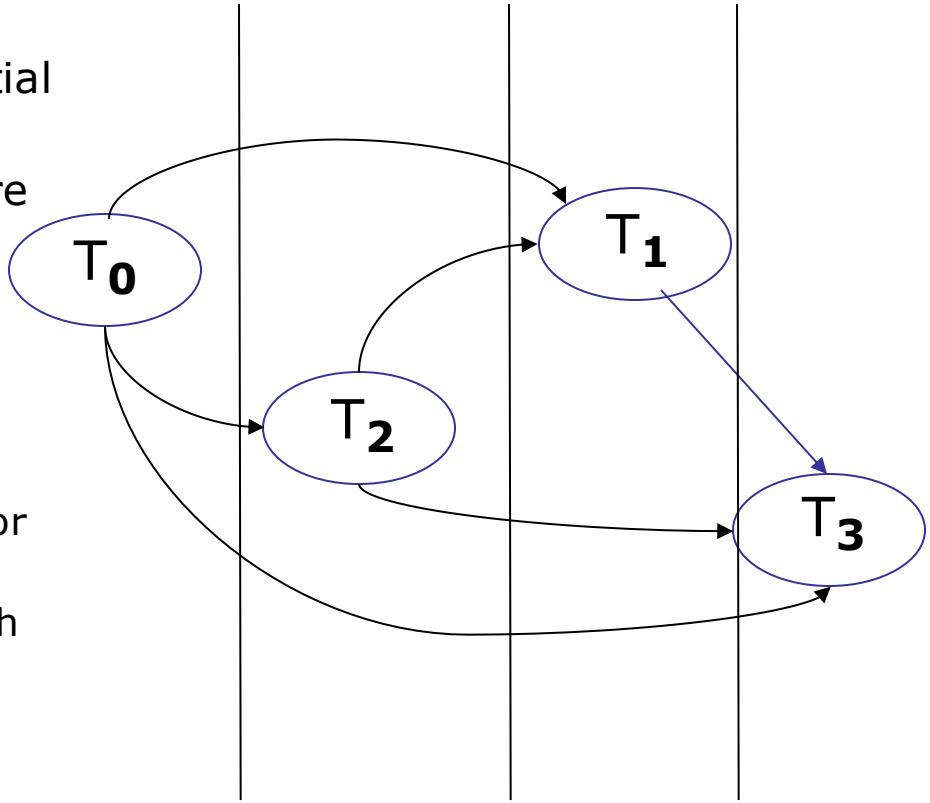


CSR implies Acyclicity of the Conflict Graph

- 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: $T_1\ T_2\ \ldots\ T_n$
- 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)
  - As many as the total orders compatible with the partial topological order



Let's go back...

- $r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$
  - it is VSR – it is view-equivalent to
    $T_1T_2T_3 = r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$
  - it is not CSR, due to the presence of the cycle between
    $r_1(x)\ w_2(x)$ and $w_2(x)\ w_1(x)$
  - there is no conflict-equivalent serial schedule, neither T1T2T3 nor T2T1T3

Try to understand the relationship between CSR and VSR...

Concurrency Control in Practice

- CSR checking would be efficient if we knew the graph from the beginning — but we don't
- A scheduler must rather work "**online**", 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
- The assumption that concurrency control can work only of the commit-projection of the schedule is unrealistic, **aborts do occur**
- Some simple on-line "decision criterion" is required for the scheduler, which must
  - **avoid as many anomalies as possible**
  - **have negligible overhead**

Arrival sequences vs a posteriori schedules

- So far the notation
  - $r_1(x)$ $w_2(x)$ $w_1(x)$ $w_3(x)$
- represented a "schedule", which is an a posteriori view of the execution of concurrent transactions in the DBMS (also called "history" in some papers and books)
- A schedule represents "what has happened", "which operations have been executed by which transaction in which order"
- They can be further restricted by the commit-projection hypothesis to operations executed by **committed** transactions
- When dealing with "online" concurrency control, it is important also to consider "**arrival sequences**", i.e., sequences of operation requests emitted in order by transactions
- With abuse of notation we will denote an arrival sequence in the same way as a posteriori schedule
  - $r_1(x)$ $w_2(x)$ $w_1(x)$ $w_3(x)$
- The distinction will be clear from the context
- The CC system maps an arrival sequence into an effective a posteriori schedule

Concurrency control approaches

- How can concurrency control be implemented "online"?
- Two main families of techniques:
  - Pessimistic
    - Based on locks, i.e., resource access control
    - If a resource is taken, make the requester wait or pre-empt the holder
  - Optimistic
    - Based on timestamps and versions
    - Serve as many requests requests as possible, possibly using out-of-date versions of the data
- We will compare the two families after introducing their features
- Commercial systems take the best of both worlds

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`** (aka SHARED LOCK) and followed by **`unlock`**
  - **`write`** operations are preceded by **`w_lock`** (aka EXCLUSIVE LOCK) and followed by **`unlock`**
- Transactions that first read and then write an object may:
  - Acquire a **`w_lock`** already when reading
  - Acquire a **`r_lock`** first and then upgrade it into a **`w_lock`** (lock escalation)
- Possible states of an object:
  - **`free`**
  - **`r-locked`** (locked by one or more readers)
  - **`w-locked`** (locked by a writer)

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

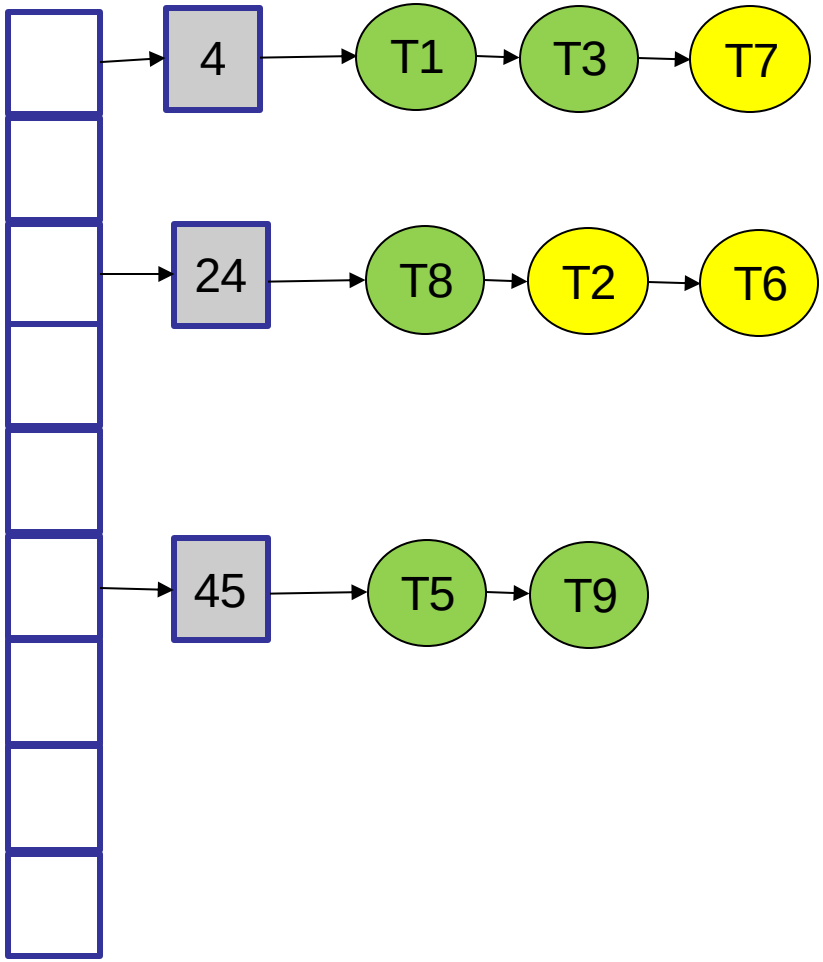
n: counter of the concurrent readers, (inc|dec)remented at each (`r_`|`un`)`lock`

Example

- r1(x)
  - r1-lock(x) request → OK → x state = r-locked, r=1
- w1(x)
  - w1-lock(x) request → OK (upgrade) → x state = w-locked
- r2(x)
  - r2-lock(x) request → NO because w-locked → T2 waits
- r3(y)
  - r3-lock(y) request → OK → y state = r-locked, T3 releases its locks → y state = free
- w1(y)
  - w1-lock(y) request → OK → y state = w-locked, T1 releases its locks → x and y states = free
    - r2(x) was waiting for r2-lock(x) request → OK x state = r-locked …
- **Arrival sequence**: r1(x), w1(x), r2(x), r3(y), w1(y)
- **A posteriori schedule**: r1(x), w1(x), r3(y), w1(y), r2(x)
- Note: the schedule depends on when T1 actually unlocks x. several a posteriori schedules are possible
- T2 is **delayed**

How are locks implemented

- Typically by lock tables, which are hash tables indexing the lockable items via hashing
- Each locked item has a linked list associated with it
- Every node in the linked list represents the transaction which requested for lock, lock mode (SL/XL) and current status (granted/waiting).
- Every new lock request for the data item is appended as a new node to the list.
- Locks can be applied on both data and index records

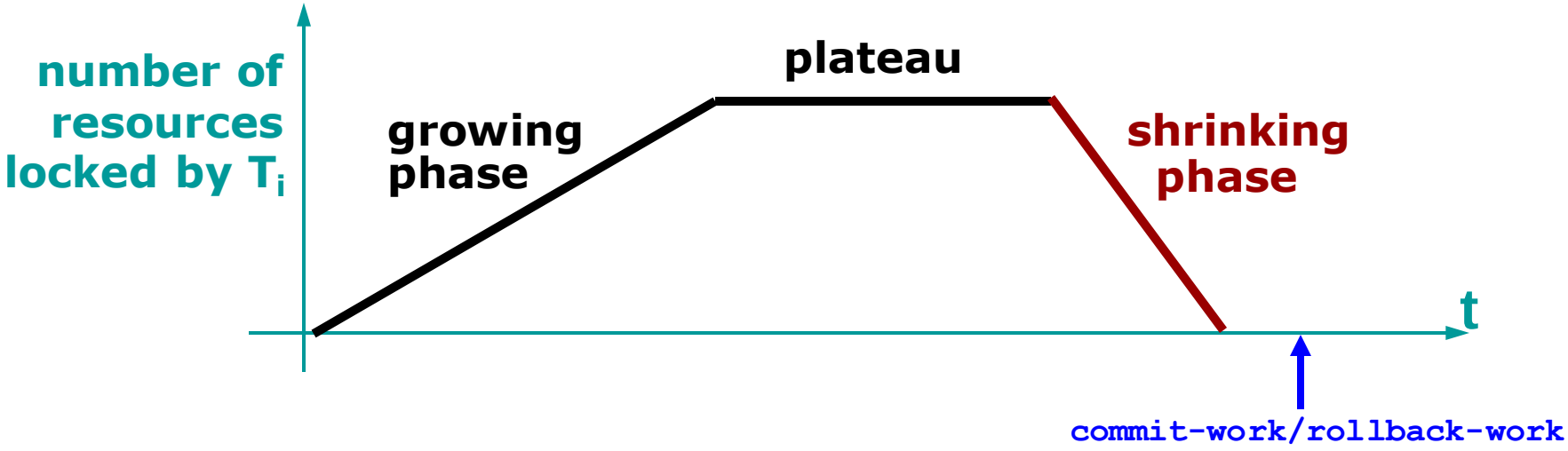


Is respecting locks enough for serializability?

- Arrival sequence r1(x), r2(x), w2(x), r1(x)
  - r1(x)
    - r1-lock(x) request → OK → x state = r-locked, r=1
      unlock_1 → x state = free
  - r2(x)
    - r2-lock(x) request → OK → x state = r-locked, r=1
  - w2(x)
    - w2-lock(x) request → OK (upgrade) → x state = w-locked
    - unlock_2 → x state = free
  - r1(x)
    - **r1-lock(x)** request → OK → x state = r-locked ..
- But the schedule mapped by the lock manager does not eliminate T1's **non repeatable read**
- T1 behavior: 1) releases its read lock on x too quickly; 2) acquires another lock after releasing one lock

Two-Phase Locking (2PL)

- Requirement (two-phase rule):
  - A transaction cannot acquire any other lock after releasing a lock
  - Sufficient to prevent non repeatable reads
  - But ensure stronger properties… serializability!



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**

Result: schedules in 2PL are view- and conflict-serializable

(VSR ⊃ CSR ⊃ 2PL)

2PL implies CSR

- CSR ⊃ 2PL: Every 2PL schedule is also conflict-serializable
- 2PL ➔ CSR:
  - Suppose a PL schedule S is not CSR
    - S graph must contain at least one cycle Ti ←→ Tj
    - Therefore there must be a pair of conflicting operations in reverse order, suppose that the conflictual operations appear as follows in the schedule
      - $OP^h_i(x)$, $OP^k_j(x)$ … $OP(y)^u_j$, $OP^w_i(y)$ where one of $OP^h_i(x)$, $OP^k_j(x)$ is a write and one of $OP(y)^u_j$, $OP^w_i(y)$ is a write
      - **$OP^h_i(x)$, $OP^k_j(x)$**
        - T_i must have released a lock for T_j to access x
      - Later in the schedule… **$OP(y)^u_j$, $OP^w_i(y)$**
        - T_i must have acquired a lock for the conflict to occur → CONTRADICTION
- Inclusion of 2PL in VSR descends from VSR ⊃ CSR
- This proves that all **2PL schedules are view-serializable**

2PL implies CSR (alternative proof)

- **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 is conflict-equivalent to S' S' $\approx_C$ S … (in all conflicting pairs transactions occur in the same order)

2PL implies CSR (alternative proof)

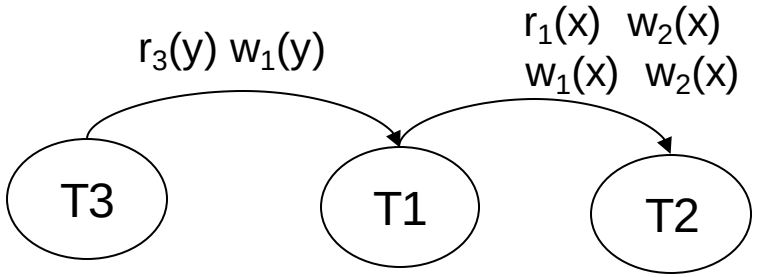
- Consider a (generic) conflict $o_i \rightarrow o_j$ in S' with $o_i \in T_i$ $o_j \in T_j$ $i<j$
- By definition of conflict, o_i and o_j address the same resource **r**, and at least one of them is a write
- Can o_i and o_j occur in reverse order in S?
  - No, because then T_j should have released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

2PL smaller than 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

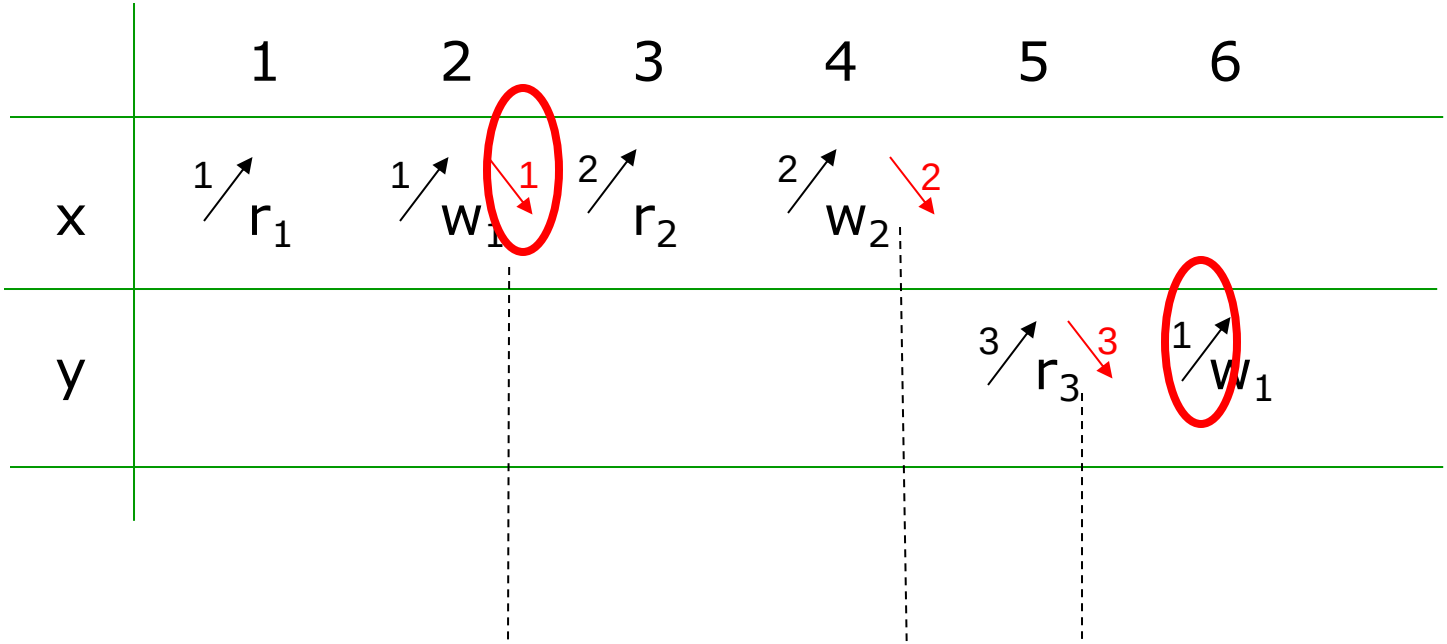
    $r_1(x)\ w_1(x)$ | $r_2(x)\ w_2(x)\ r_3(y)$ | $w_1(y)$

    T_1 releases **T_1 acquires**
  - However, it is conflict-serializable: $T_3 < T_1 < T_2$
    - On x: $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)$
    - On y: $r_3(y)\ w_1(y)$



A visualization of the 2PL test

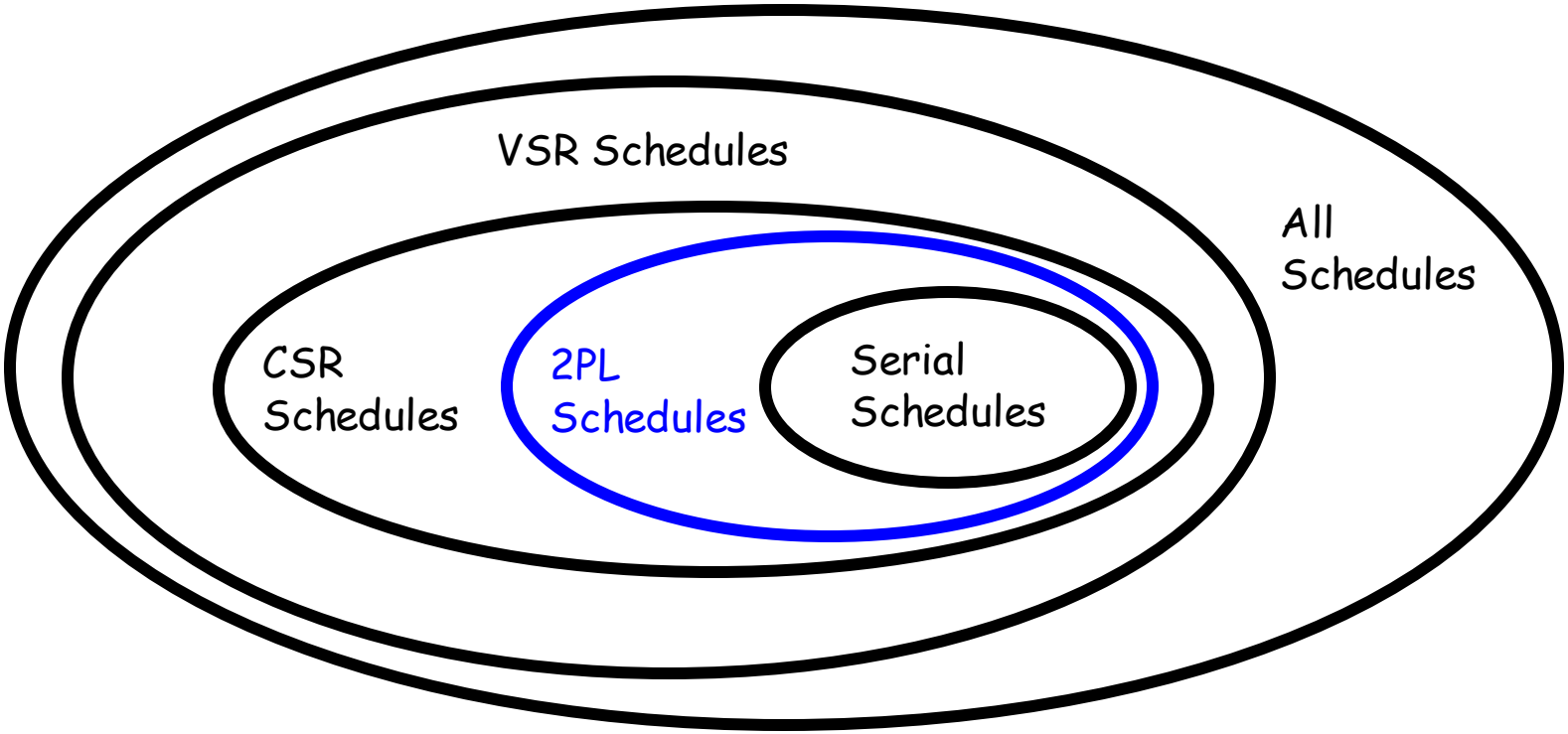
Resources on the Y axis and operation times on the X axis



2PL and other anomalies

- Nonrepeatable read r1 - r2 - w2 - r1
  - Already shown
- Lost update r1 - r2 - w2 - w1
  - T1 releases a lock to T2 and then tries to acquire another one
- Phantom update: r1 - r2 - w2 - r1
  - T1 releases a lock to T2 and then tries to acquire another one
- Phantom insert: r1 - w2(new data) - r1
  - releases a lock to T2 and then tries to acquire another one (requires lock on "future data" aka predicate locks)
- Dirty read r1 - w1 - r2 - abort1 - w2
  - **Requires dealing with abort**

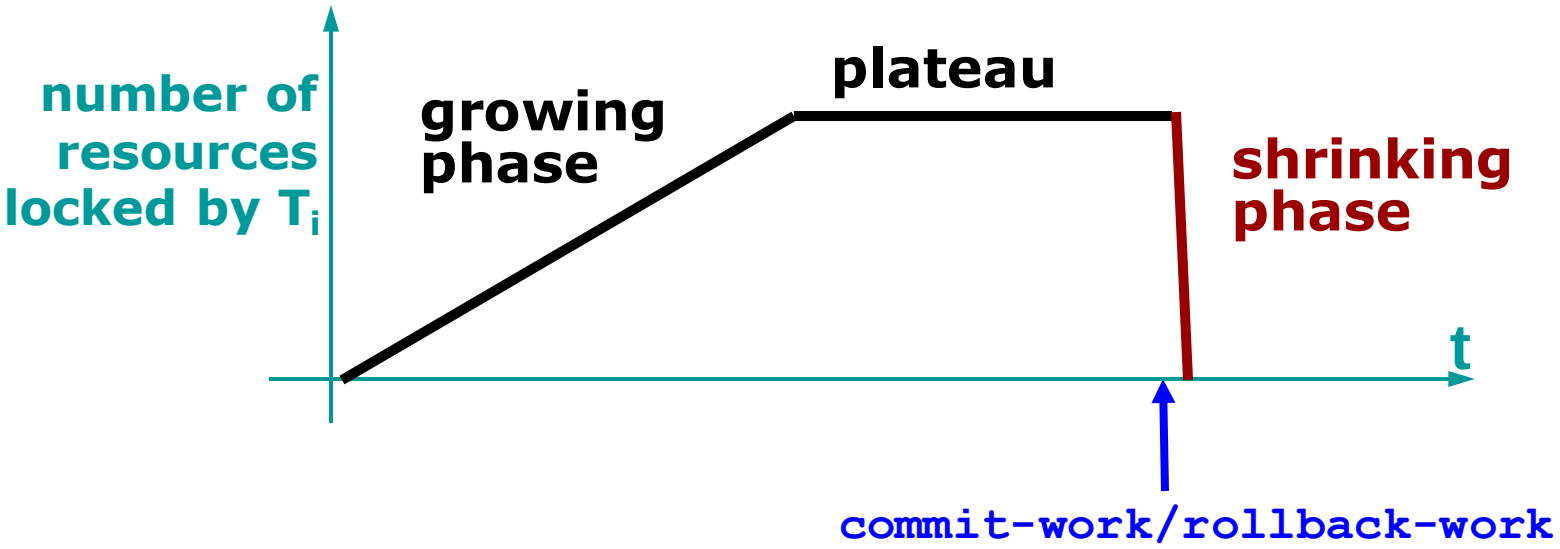
CSR, VSR and 2PL



Dirty reads are still a menace: Strict 2PL

- **Up to now, we were still using the hypothesis of *commit-projection* (no transactions in the schedule abort)**
  - 2PL, as seen so far, does not protect against dirty (uncommitted data) 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*
  - *Remember: rollback restores the state prior to the aborted updates*
- This version of 2PL is used in most commercial DBMSs when a high level of isolation is required (see next: SQL isolation levels)

Strict 2PL in Practice



- Strict 2PL locks are also called **long duration** locks, 2PL locks **short duration** locks
- Note: real systems may apply 2PL policies differently to read and write locks
- Typically: long duration, strict 2PL write locks, variable policies for read locks

How to prevent phantom inserts: predicate locks

- A phantom insertion occurs when a transaction adds items to a data set previously read by another transaction
- To prevent phantom inserts a lock should be placed also on "future data", i.e., inserted data that **would satisfy** a previous query
- **Predicate locks** extend the notion of **data locks** to "future data"
- 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 managed with the help of indexes (gap lock)

Tab

A	B

Isolation Levels in SQL:1999 (and JDBC)

- SQL defines **transaction isolation levels** which specify the anomalies that should be prevented by running at that level
- The level does not affect write locks. A transaction should always get exclusive lock on any data it modifies, and **hold it until completion (strict 2P on write locks)**, regardless of the isolation level. For read operations, levels define the degree of protection from the effects of modifications made by other transactions

Why long duration write locks are necessary

- Consider the following schedule (admissible if write locks are short duration) and **remove the hypothesis that aborts do not occur**
- w1[x]...w2[x]...((c1 or **a1**) and (c2 or **a2**) in any order)
  - T2 is allowed to write over the same object updated by T1 which has not yet completed
- If T1 aborts.. e.g., w1[x]...w2[x] ... , **a1**, (c2 or a2)
  - How to process event a1?
    - If x is restored to the state before T1, T2's update is lost, so if T2 commits x has a stale value
    - If x is NOT restored and T2 also aborts.. then T2's proper before state cannot be reinstalled either!
- Thus: write locks are held until the completion of the transaction to enable the proper processing of abort events
- The anomaly of the above non commit-projection schedule is named **dirty write**

Isolation Levels in SQL:1999 (and JDBC)

- `READ UNCOMMITTED` allows dirty reads, nonrepeatable reads and phantom updates and inserts:
  - No read locks (and **ignores** locks of other transactions)
- `READ COMMITTED` prevents dirty reads but allows nonrepeatable reads and phantom updates/inserts:
  - Read locks (and complies with locks of other transactions), but without 2PL **on read locks** (read locks are released as soon as the read operation is performed and can be acquired again)
- `REPEATABLE READ` avoids dirty reads, nonrepeatable reads and phantom updates, but allows phantom inserts:
  - long duration read locks → 2PL also for reads
- `SERIALIZABLE` avoids all anomalies:
  - 2PL with predicate locks to avoid phantom inserts
- Note that SQL standard isolation levels dictate minimum requirements, real systems may go beyond (e.g., in MySQL and Postgres REPEATABLE READ avoids phantom inserts too)

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

SQL92 serializable != serial!

- Serializable transactions don't necessarily execute serially
- The requirement is that transactions can only commit if the result would be as if they had executed serially in any order
- The locking requirements to meet this guarantee can frequently lead to deadlock where one of the transactions needs to be rolled back
- Therefore SERIALIZABLE isolation level is used sparingly and in NOT the default in most commercial systems

SQL isolation levels and locks

- SQL Isolation levels may be implemented with the appropriate use of lock
- Commercial systems make joint use of locks and of timestamp-based concurrency control mechanisms

	READ LOCKS	WRITE LOCKS
READ UNCOMMITTED	Not required	Well formed writes Long duration rite locks
READ COMMITTED	Well formed reads Short duration read locks (data and predicate)	Well formed writes Long duration rite locks
REPEATABLE READ	Well formed reads Long duration data read locks Short duration predicate read locks	Well formed writes Long duration rite locks
SERIALIZABLE	Well formed reads Long duration read locks (predicate and data)	Well formed writes Long duration rite locks

Setting transaction characteristics in SQL

```
<set transaction statement> ::=
SET [ LOCAL ] TRANSACTION <transaction characteristics>

<transaction characteristics> ::= [ <transaction mode> [ { <comma>
  <transaction mode> }... ] ]

<transaction mode> ::= <isolation level> | <transaction access mode>
  | <diagnostics size>

<transaction access mode> ::= READ ONLY| READ WRITE

<isolation level> ::= ISOLATION LEVEL <level of isolation>

<level of isolation> ::= READ UNCOMMITTED | READ COMMITTED
  | REPEATABLE READ | SERIALIZABLE
```

The impact of Locking: waiting is dangerous!

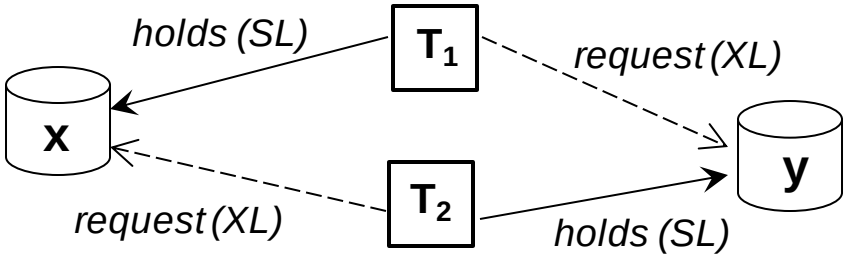
- Locks are tracked by lock tables (main memory data structures)
  - Resources can be Free, or Read-Locked, or Write-locked
  - To keep track of readers, every resource also has a "read counter"
- Transactions requesting locks are either **granted the lock** or **suspended and queued** (first-in first-out). 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: r1 r2 w1 w2 → 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

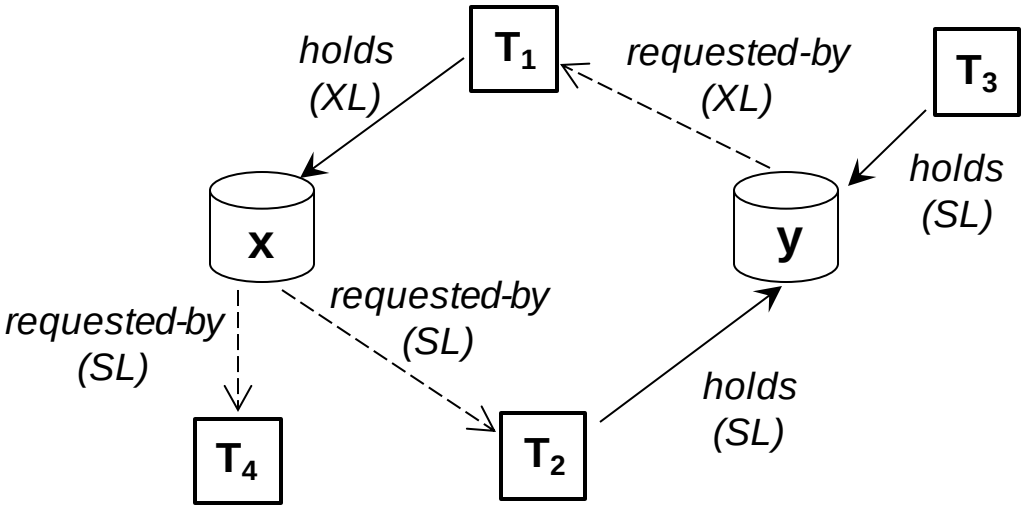
- T_1 : $r_1(x)$ $w_1(y)$
- T_2 : $r_2(y)$ $w_2(x)$

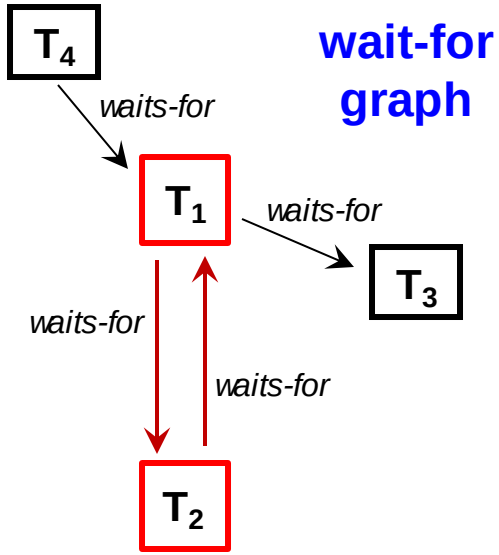
S: $r\_lock_1(x)$ $r\_lock_2(y)$ $r_1(x)$ $r_2(y)$ $w\_lock_1(y)$ $w\_lock_2(x)$



Deadlock

- Lock graph: a bipartite graph in which nodes are resources or transactions and arcs are lock requests or lock assignments
- Wait for graph: a graph in which nodes are transactions and arcs are "waits for" relationships
- A deadlock is represented by a cycle in the wait-for graph of transactions



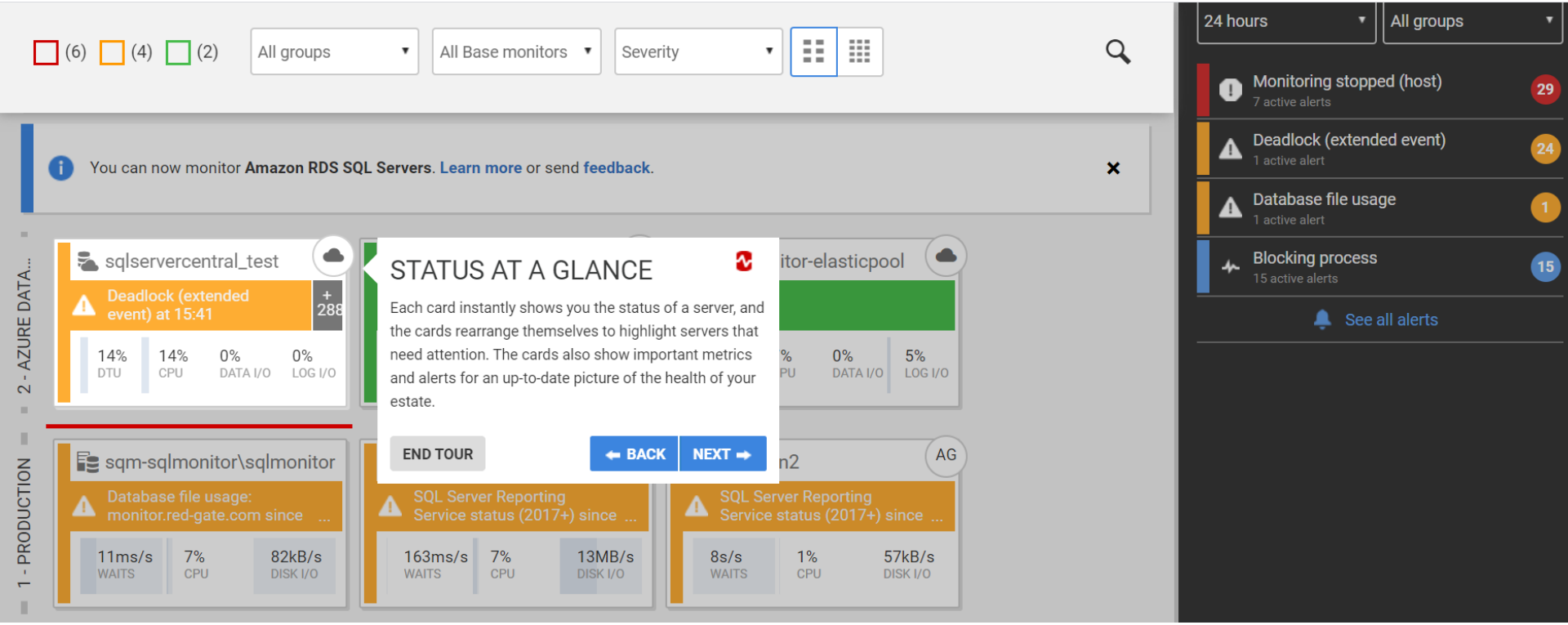


Deadlock Resolution Techniques

- 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

Concurrency Control

Deadlock detection in commercial dashboards



Timeout Method

- A transaction is killed and restarted 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

http://davebland.com/how-often-does-sql-server-look-for-deadlocks

Deadlock Prevention

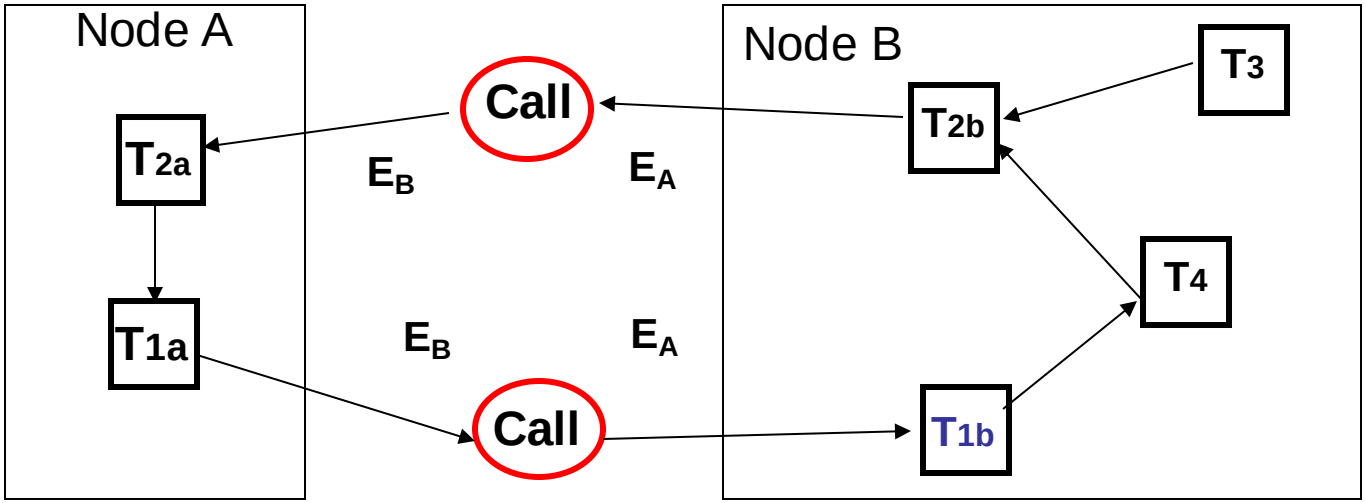
- Idea: killing transactions that could cause cycles
- **Resource-based prevention**: Restrictions on lock requests
  - Transactions request all resources at once, and only once
  - Resources are globally sorted and must be requested "in global order"
  - Problem: it's not easy for transactions to anticipate all requests!
- **Transaction-based prevention**: restrictions based on transactions' IDs
  - Assigning IDs to transactions incrementally → transactions' "age"
  - Preventing "older" transactions from waiting for "younger" ones to end their work
  - Options for choosing the transaction to kill
    - **Preemptive** (killing the holding transaction – wound-wait)
    - **Non-preemptive** (killing the requesting transaction – wait-die)
  - Problem: 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 efficiently & reliably
- An elegant solution: **Obermark**'s algorithm (DB2-IBM, published on ACM-Transactions on Database Systems)
- Assumptions
  - Transactions execute on a single main node (one locus of control)
  - Transactions may be decomposed in "sub-transactions" running on other nodes
  - Synchronicity: when a transaction spawns a sub-transaction it suspends work until the latter completes
  - Two waits-for relationships:
    - T_i waits from T_j on the same node because T_i needs a datum locked by T_j
    - A sub-transaction of T_i waits for another sub-transaction of T_i running on a different node (via external call E)

Distributed Deadlock Detection: Problem Setting

Distributed dependency graph: external call nodes represent a sub-transaction activating another sub-transaction at a different node

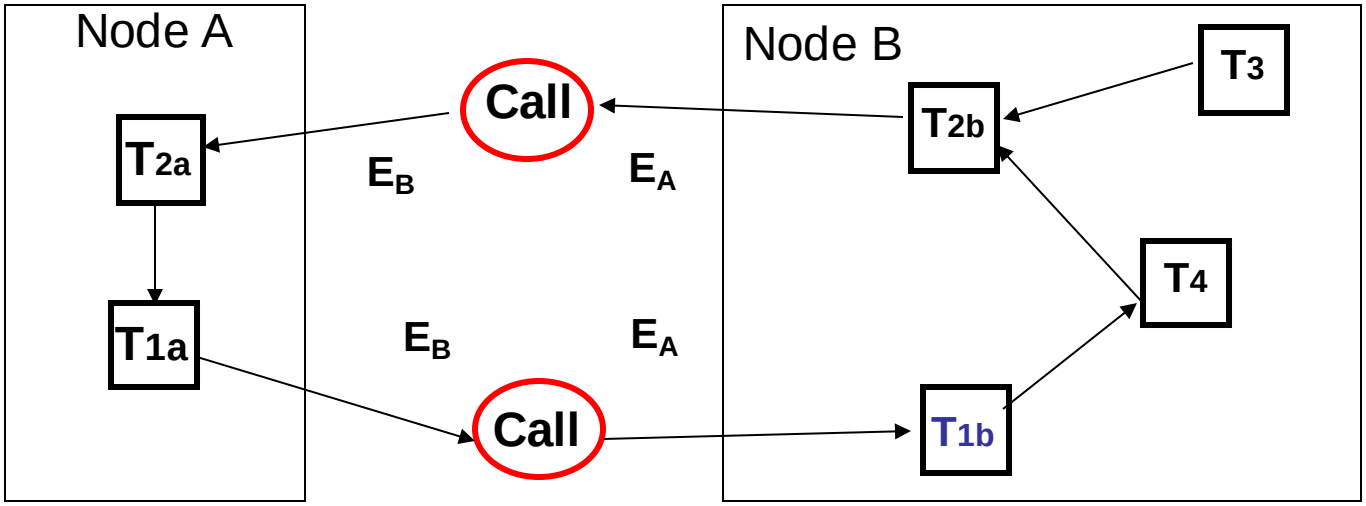


- Representation of the status
  - at Node A: $E_b \rightarrow T_{2a} \rightarrow T_{1a} \rightarrow E_b$
  - at Node B: $E_a \rightarrow T_{1b} \rightarrow T_{2b} \rightarrow E_a$
- NOTATION: symbol → : 1) «wait for» relation among local transactions 2) if one term is an external call → the source is waited for by an remote transaction or the sink waits for a remote transcation
- Potential Deadlock: T_{2a} waits for T_{1a} (data lock) that waits for T_{1b} (call) that waits for T_{2b} (data locks) that waits for T_{2a} (call): cycle!
- Problem: how to detect such occurrence without maintaining the global view

Obermark's Algorithm

- Goal: detection a potential deadlock looking only at the local view of a node
- Method: establishing a communication protocol whereby each node has a local projection of the global dependencies
- Nodes exchange information and update their local graph based on the received information
- Communication is optimized to avoid that multiple nodes detect the same potential deadlock
- Node A sends its local info to a node B only if
  - A contains a transaction T_i that is waited for from another remote transaction and waits for a transaction T_j active on B
  - i>j (this ensure a kind of message «forwarding» along a node path where node A «precedes» node B if i<j)
  - Mnemonically: I send info to you if a d-transaction listed at me waits for a d-transaction listed at you with smaller index

Distributed Deadlock Detection: forwarding rule

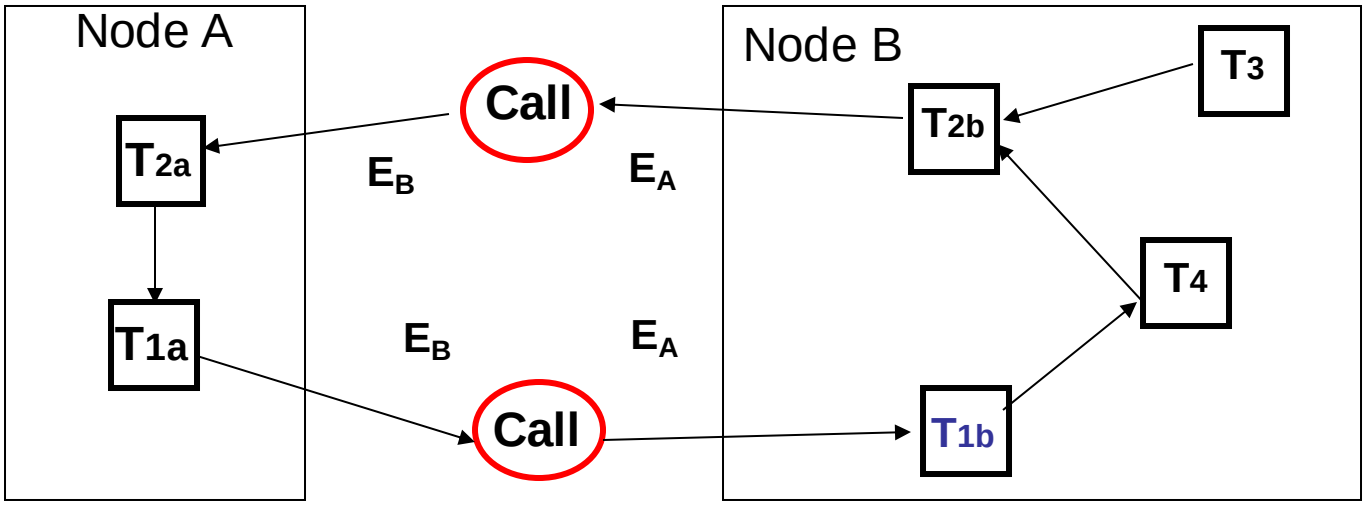


- Node A:
  - Activation/wait sequence: $E_b \rightarrow T_2 \rightarrow T_1 \rightarrow E_b$
    - i=2, j=1
    - A can dispatch info to B
- Node B:
    - Activation/wait sequence (only distributed transactions count): $E_a \rightarrow T_1 \rightarrow T_2 \rightarrow E_a$
    - i=1, j=2
    - B does not dispatch info to A

Obermark's Algorithm

- Runs periodically at each node
- Consists of 4 steps
  1. *Get graph info (wait dependencies among transactions and external calls) from the "previous" nodes. Sequences contain only node and top-level transaction identifiers*
  2. *Update the local graph by merging the received information*
  3. *Check the existence of cycles among transactions denoting potential deadlocks: if found, select one transaction in the cycle and kill it*
  4. *Send updated graph info to the "next" nodes*

Algorithm execution, step 1: communication



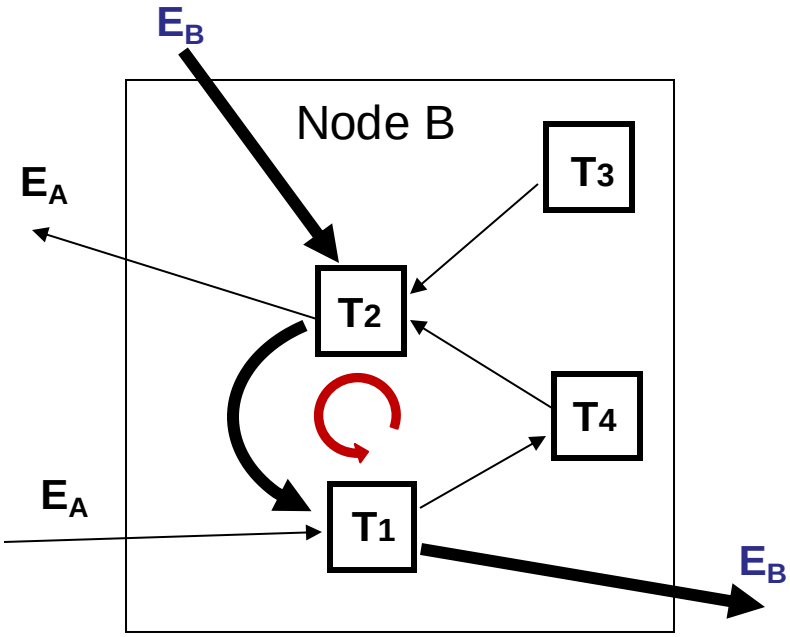
- Distributed Deadlock Detection: forwarding rule
  - at Node A: $E_b \rightarrow T_2 \rightarrow T_1 \rightarrow E_b$ info sent to Node B
  - at Node B: $E_a \rightarrow T_1 \rightarrow T_2 \rightarrow E_a$ info not sent (i<j)

Algorithm execution, step 2: local graph update

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 (cycle among T1 and T2:

   T_1 or T_2 or T_4 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, $E_B \rightarrow T_1 \rightarrow T_3 \rightarrow E_A$
- Node A: 3 > 2 can send (to B)
- Node B: 2 > 1 can send (to C)
- Node C: 1 < 3 cannot send
- We assume that node A is the first to execute, sending its sequence to B (i.e., to the node hosting the (sub)transaction that keeps A's transaction on hold)

Another example, continued

- At node B:
- $E_A \rightarrow T_2 \rightarrow T_1 \rightarrow E_C$ is merged with $E_C \rightarrow T_3 \rightarrow T_2 \rightarrow E_B$
- Result:

$E_C \rightarrow T_3$ ↘

$E_A \rightarrow T_2 \rightarrow T_1 \rightarrow E_C$

↘ E_B

- A new potential deadlock $E_C \rightarrow T_3 \rightarrow T_1 \rightarrow E_C$ is found by combining the $E_C \rightarrow T_3 \rightarrow T_2 \rightarrow E_B$ message with the local info $E_A \rightarrow T_2 \rightarrow T_1 \rightarrow E_C$. This can be sent to node C (3>1)
- At node C:
- Local info $E_B \rightarrow T_1 \rightarrow T_3 \rightarrow E_A$ is merged with message $E_C \rightarrow T_3 \rightarrow T_1 \rightarrow E_C$

$$E_B \rightarrow T_1 \leftarrow \rightarrow T_3 \rightarrow E_A$$

- The deadlock 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)

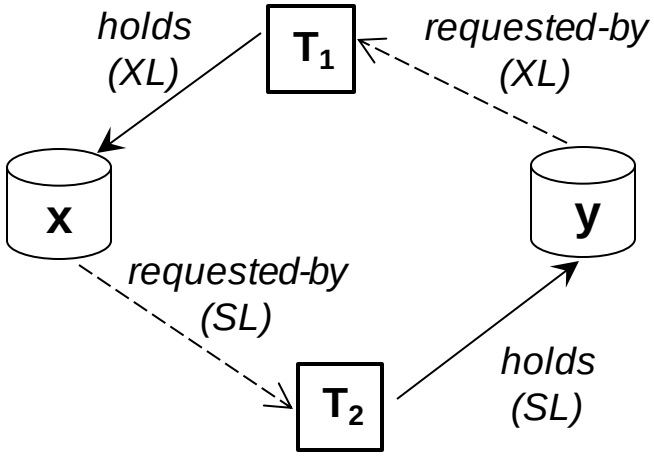
Deadlocks in practice

- 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(**$1/n^2$**)
- 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 resources (SL) and then decide to write 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

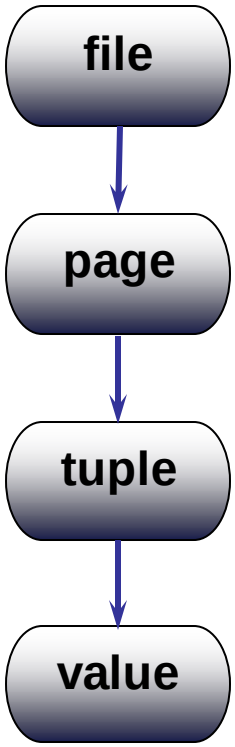
	Resource status		
Request	SL	UL	XL
SL	OK	**OK**	No
UL	**OK**	**No**	No
XL	No	No	No



- Update locks are easy to implement and mitigate the most frequent cause of collision r1,r2,w1,w2
- They are requested by using SQL SELECT **FOR UPDATE** statement

Hierarchical Locking

- Update locks prudentially extend the interval during which a resource is locked
- What to lock? An entire table? → reduces concurrency too much
- Locks can be specified with different granularities
  - e.g.: schema, table, fragment, page, tuple, field
- Objectives:
  - Locking the minimum amount of data
  - Recognizing conflicts as soon as possible
- Method: asking locks on hierarchical resouces by:
  - Requesting resources top-down until the right level is obtained
  - Releasing locks bottom-up



Reduced Granularity

Increased Concurrency

Intention Locking Scheme

- 5 Lock modes:
  - In addition to read (SHARED) locks (**SL**) and write (EXCLUSIVE) locks (**XL**)
- The new modes express the "**intention** of locking at lower (finer) levels of granularity"
  - **ISL**: Intention of locking a subelement of current element in shared mode
  - **IXL**: Intention of locking a subelement of current element in exclusive mode
  - **SIXL**: Lock of the element in shared mode with intention of locking a subelement in exclusive mode (SL+IXL)

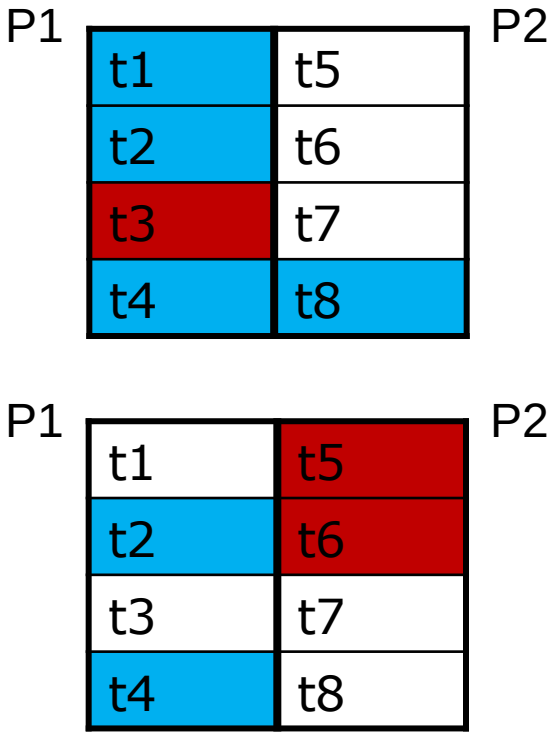
Hierarchical Locking Protocol

1. Locks are requested starting from the **root** (e.g., starting from the whole table) and going down in the hierarchy
2. Locks are released starting from the leaves and going up in the hierarchy
3. To request an SL or ISL lock on a non-root element, a transaction must hold an equally or more restrictive lock (ISL or IXL) on its "parent"
4. To request an IXL, XL or SIXL lock on a non-root element, a transaction must hold an equally or more restrictive lock (SIXL or IXL) on its "parent"
5. When a lock is requested on a resource, the lock manager decides based on the rules specified in the hierarchical lock granting table

Hierarchical lock granting table

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

Example



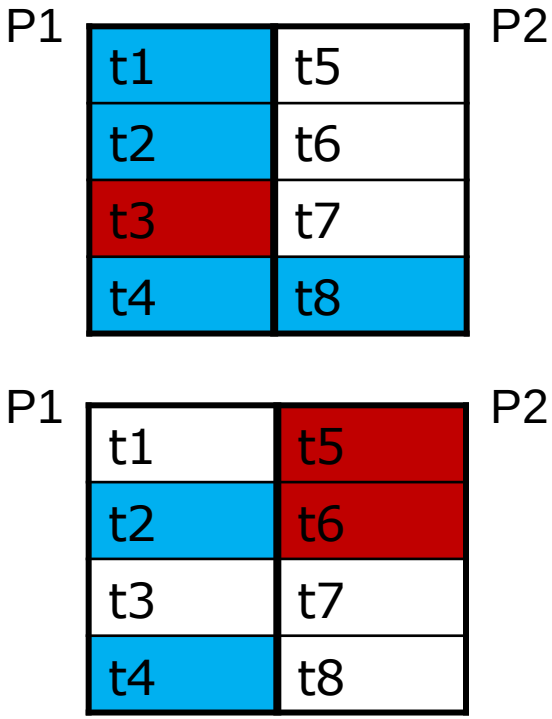
Root = TableX
Page 1 (P1): t1,t2,t3,t4 tuples
Page 2 (P2): t5,t6,t7,t8 tuples

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

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

They are NOT in r-w 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)

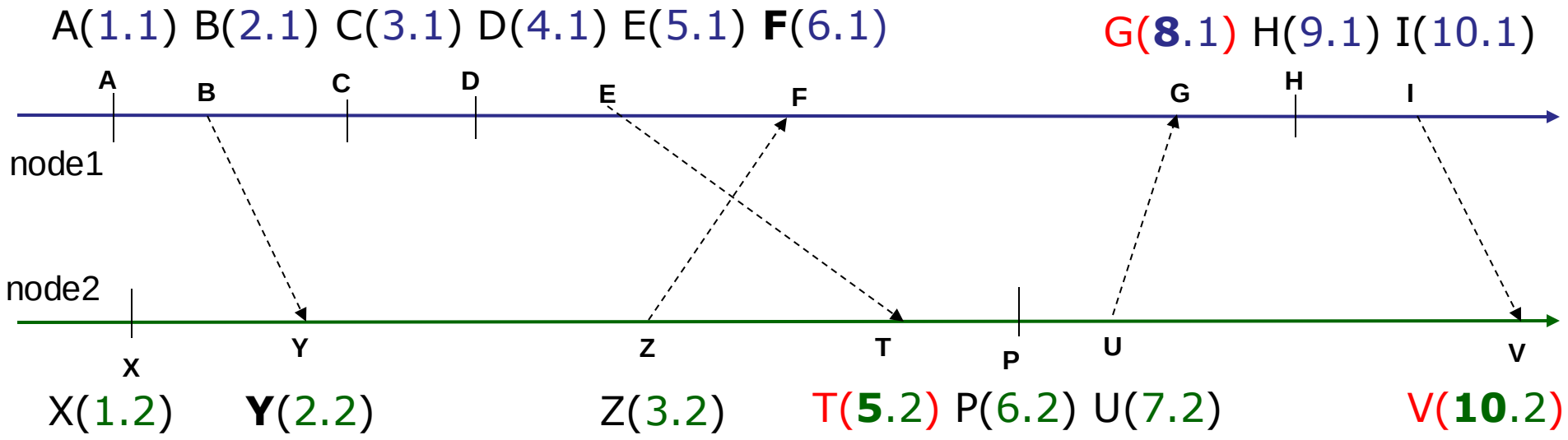
Concurrency Control Based on Timestamps

- Locking is also named **pessimistic** concurrency control because it assumes that collisions (transactions reading-writing the same object concurrently) will arise
- In reality collisions are rare
- Alternative to 2PL (and to locking in general) are **optimistic** concurrency control methods
- **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 so that transactions can be ordered by "birth date"
- A schedule is accepted only if it reflects the serial ordering of the transactions induced by their timestamps

Assigning timestamps in distributed systems

- 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
- Note that the notion of time is "lexical": timestamp 5.1 "occurs before" timestamp 5.2
- Synchronization: send-receive of messages
  - for a given message m, send(m) precedes receive(m)
- Algorithm (Lamport method): cannot receive a message from "the future", if this happens the "bumping rule" is used to bump the timestamp of the *receive* event beyond the timestamp of the *send* event
- Mnemonically: if I receive a message from you that has a timestamp greater than my last emitted one I update my current timestamp to exceed yours

Example of timestamp assignment



- Events **Y** and **F** represent messages received "from the present or past" → OK
- Events T, G and V represent messages received "from the future" → The local timestamp is incremented (*bumped*) accordingly
- Note that the timestamp of the receive event T, G and V is generated so to exceed that of the send event

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
- Note: TS-based control considers only committed transactions in the schedule, aborted transactions are not considered (commit-projection hypothesis)
- To work w/o the commit-projection hypothesis, write operations should be buffered until commit, in this way aborted transactions do not even issue write requests and the commit-projection hypothesis is equivalent to a full schedule with committed and aborted transactions
- But…buffering introduces delays

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

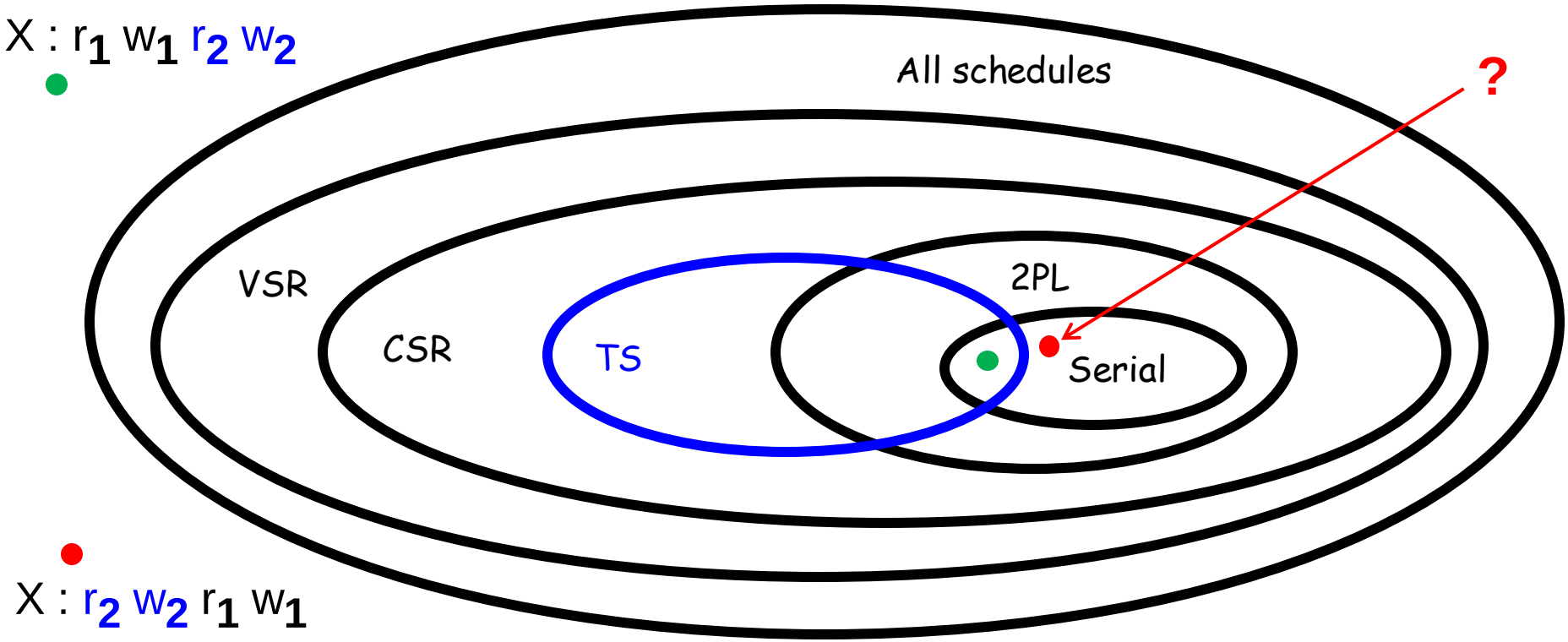
- They are incomparable
  - Schedule in TS but not in 2PL

    $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_0(y)\ w_1(y)$
  - 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



2PL vs. TS

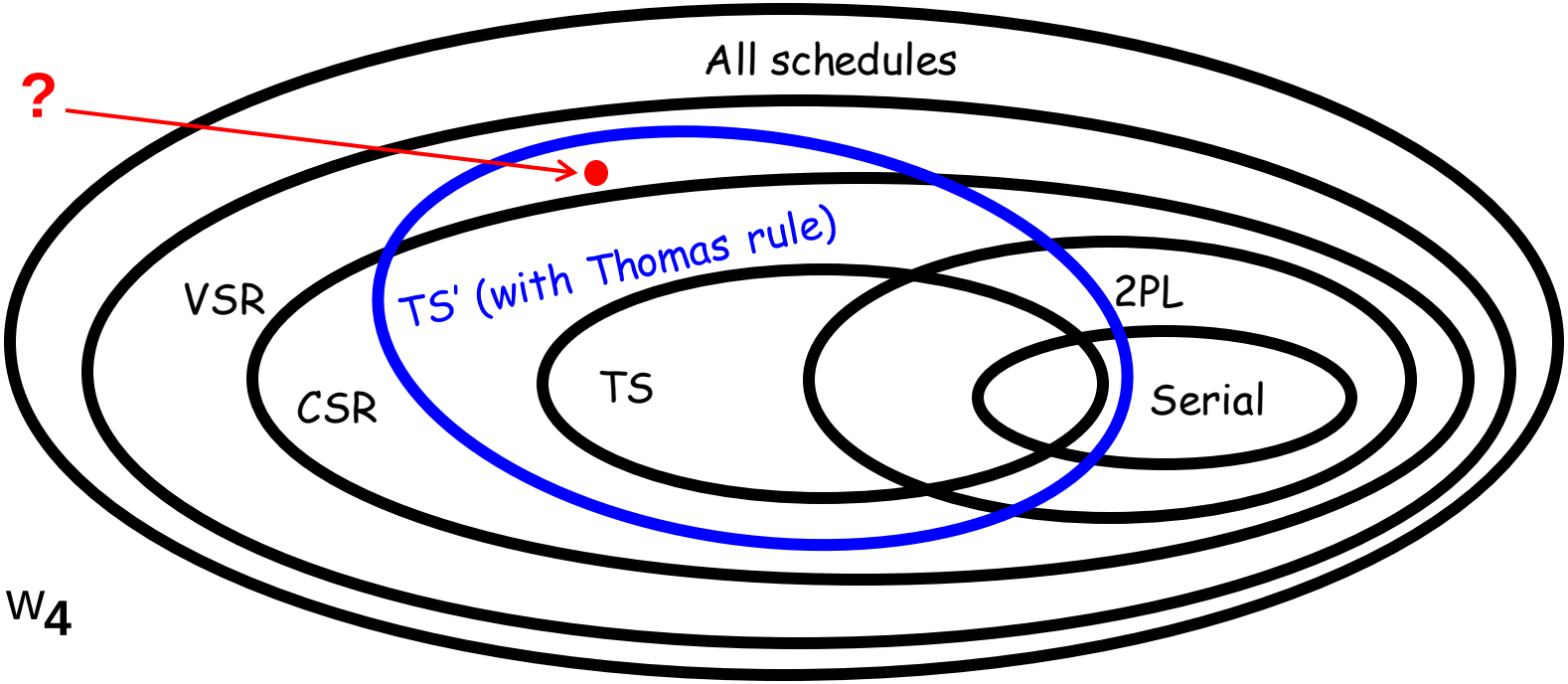
- In *2PL* transactions can be actively waiting. In *TS* they are killed and restarted
- The serialization order with *2PL* is imposed by conflicts, while in *TS* it is imposed by the timestamps
- The necessity of waiting for commit of transactions causes long delays in *strict 2PL*
- *2PL* can cause deadlocks
  - but also *TS,* if care is not taken
- Restarting a transaction costs more than waiting: *2PL* wins!
- Commercial systems implement a mix of optimistic and pessimistic concurrency control (e.g., Strict 2PL or 2PL + Multi Version TS)

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

- Rationale: skipping a write on an object that has already been written by a younger transaction, without killing the transaction
- Works only if the transaction issues a write without requiring a previous read on the object (so SET X = X+1 would fail)
- 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

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

$w_2(y)$ is skipped

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

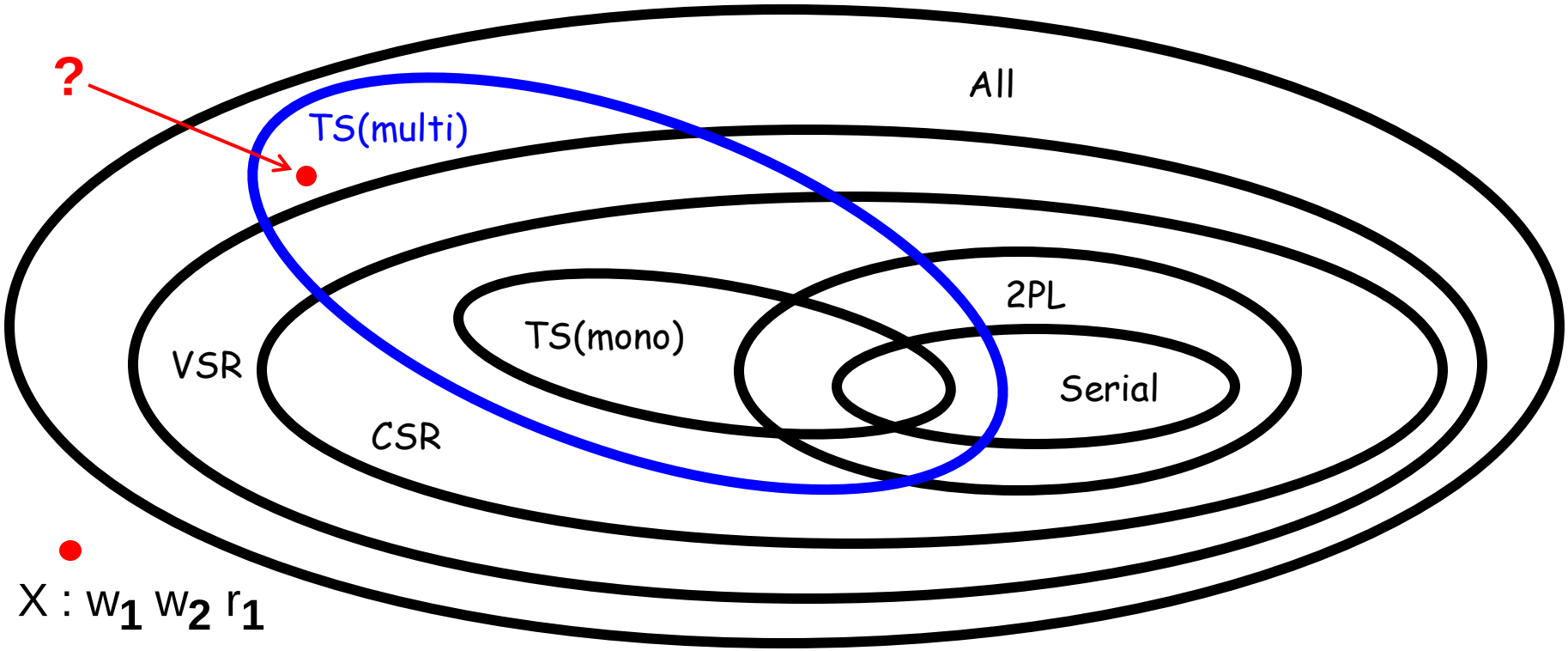
- Mechanism:
  - *read*(x,ts) is always accepted. A copy x_k is selected for reading such that:
    - If $ts > WTM_N(x)$, then $k = N$
    - Else take k such that $WTM_k(x)$ **<= ts** $< WTM_{k+1}(x)$
  - *write*(x,ts):
    - If $ts < RTM(x)$ the request is **rejected**
    - Else a new version is created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



- Versions: X0 (original value), X1, X2
- $r_1$reads X1 [$WTM_k(x)$ <= ts < $WTM_{k+1}(x)$]
- Schedule not in VSR
  - T1, T2 different final write
  - T2, T1 different reads from for $r_1(x)$

Snapshot Isolation (SI)

- The realization of multi-TS gives the opportunity to introduce into DBMSs 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 (i.e., the version that existed when the transaction started a.k.a. snapshot), and defers writes to the end
- If a transaction notices that its writes damage writes made by other concurrent transactions after the snapshot, it aborts
  - It is yet another case of optimistic concurrency control

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

Concurrency control in some commercial DBMS

- ORACLE

https://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm

- MYSQL

https://dev.mysql.com/doc/refman/8.0/en/locking-issues.html

- IBM DB2

https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.7.0/com.ibm.db2.luw.admin.perf.doc/doc/c0054923.html

- MONGODB

https://docs.mongodb.com/manual/faq/concurrency/