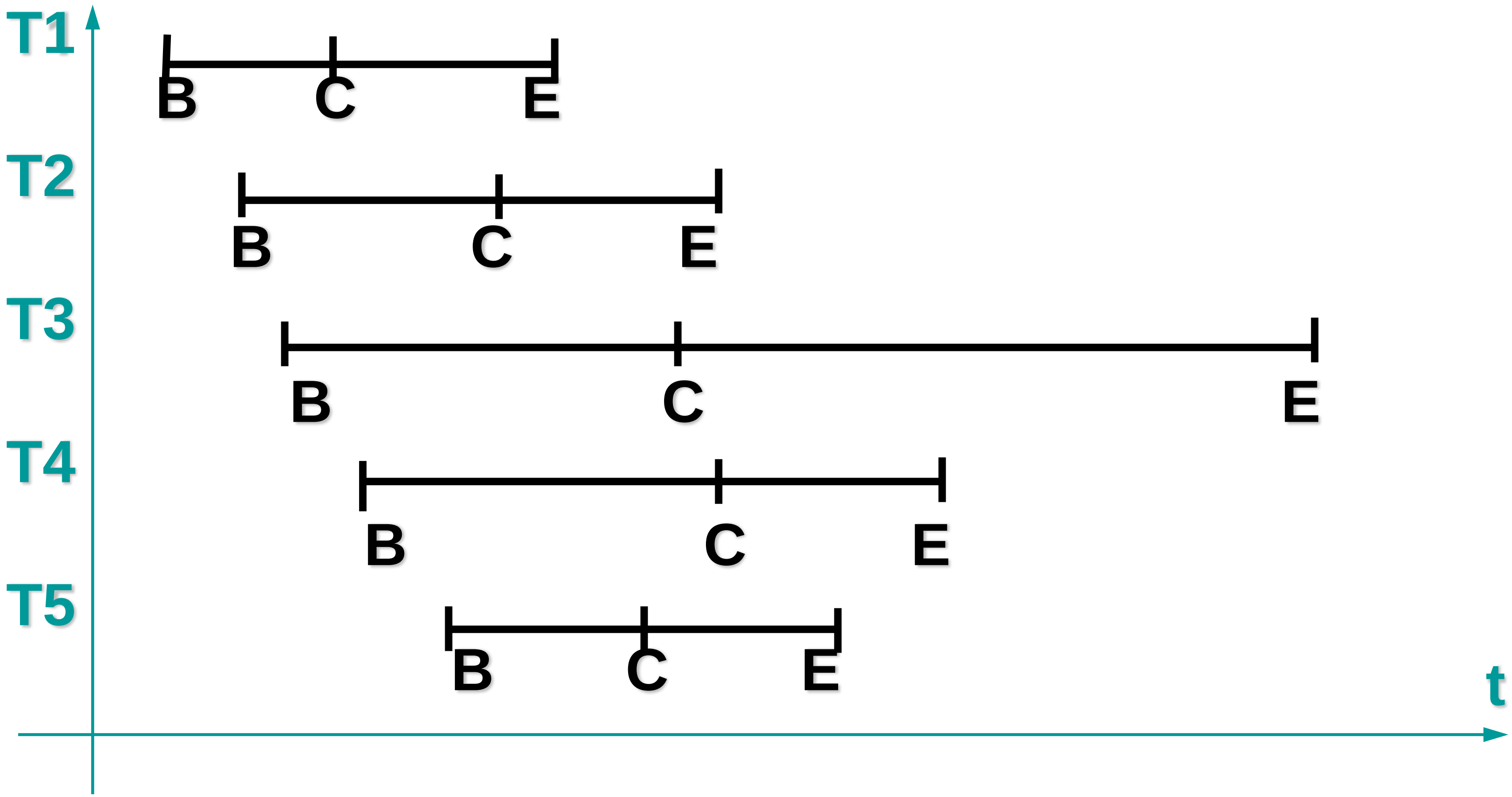
Advanced Databases

2 Concurrency Control

Advantages of Concurrency



Two concurrent transactions

T1 :
```
begin transaction
UPDATE account
      SET balance = balance + 3
      WHERE client = 'Smith'
commit work
end transaction
```

T2 :
```
begin transaction
UPDATE account
      SET balance = balance + 6
      WHERE client = 'Smith'
commit work
end transaction
```

Low level view of the transactions/1

T1 :
```
begin transaction
X:= X + 3
commit work
end transaction
```

T2 :
```
begin transaction
X:= X + 6
commit work
end transaction
```

Low level view of the transactions/2

T1 :
```
begin transaction
read(X,v1)
v1:= v1 + 3
write(v1,X)
commit work
end transaction
```

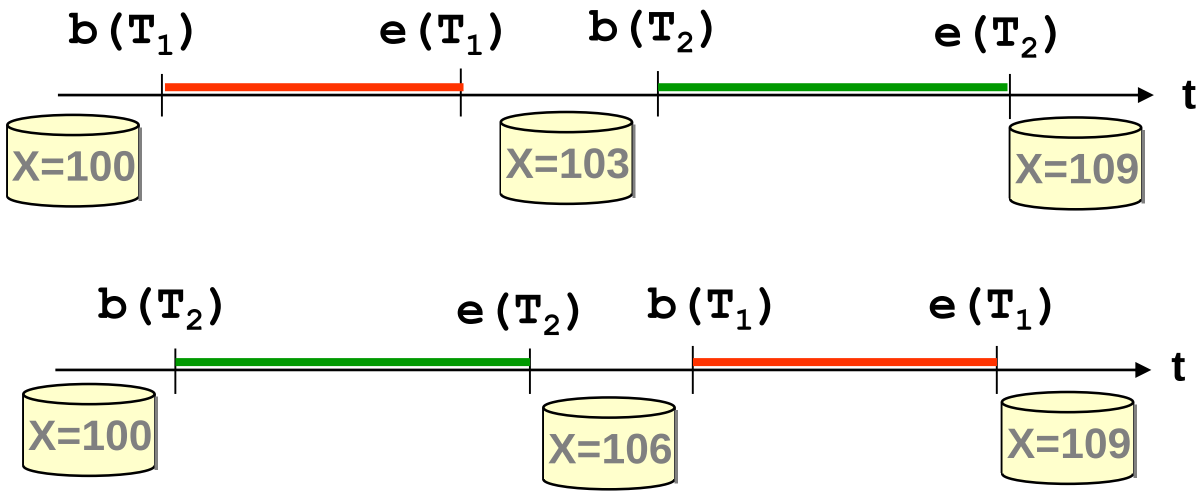
T2 :
```
begin transaction
read(X,v2)
v2:= v2 + 6
write(v2,X)
commit work
end transaction
```

Serial Executions

```
bot(T1)
    r(X,v1)
    v1=v1+3
    w(v1,X)
    commit
eot(T1)
```

```
bot(T2)
    r(X,v2)
    v2=v2+6
    w(v2,X)
    commit
eot(T2)
```

X_0=100

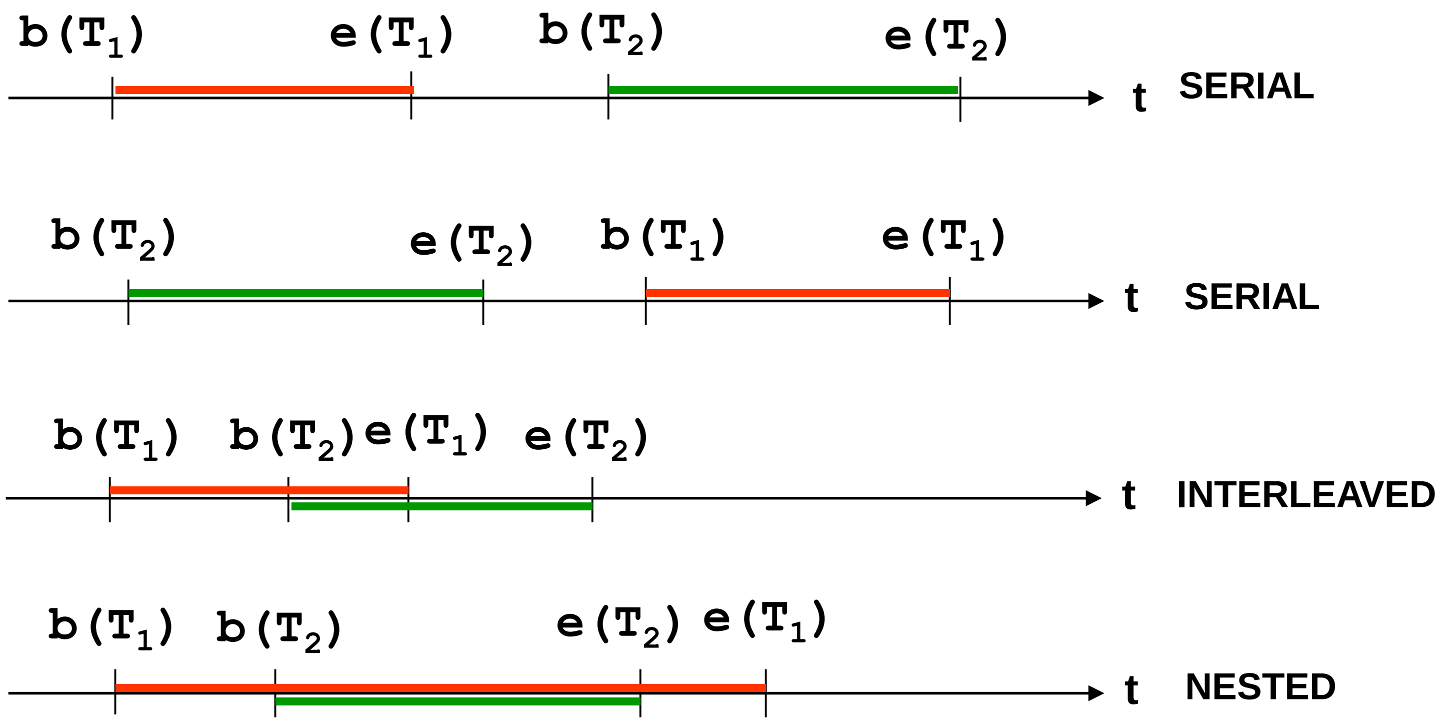


Concurrency Control

- Concurrency is fundamental
  - Tens, hundreds, thousands 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



Problems due to Concurrency

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

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

Execution with Lost Update

X=100

1 T1: R(X,V1)
2 V1 = V1 + 3
3 T2: R(X,V2)
4 V2 = V2 + 6
5 T1: W(V1,X) X=103
6 T2: W(V2,X) X=106!

Sequence of I/O Actions producing the Error

R(T1) R(T2) W(T1) W(T2)

or

R(T1) R(T2) W(T2) W(T1)

“Dirty” Read

X=100

1 T1: R(X,V1)
2 T1: V1 = V1 + 3
3 T1: W(V1,X) X=103
4 T2: R(X,V2)
5 T1: ROLLBACK
6 T2: V2 = V2 + 6
7 T2: W(V2,X) X=109!

“Nonrepeatable” Read

X=100

1 T1: R(X,V1)
2 T2: R(X,V2)
3 T2: V2 = V2 + 6
4 T2: W(V2,X) X=106
5 T1: R(X,V3) V3<>V1!

Ghost Update

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

T1: R(X,V1), R(Y,V2)
T2: R(Y,V3), R(Z,V4)
T2: V3 = V3 + 10, V4=V4-10
T2: W(V3,Y), W(V4,Z)   (Y=40, Z=10)
T1: R(Z,V5)   (for T1, V1+V2+V5=90!)

Phantom Insert

T1: C=AVG(B: A=1)

T2: Insert (A=1,B=2)

T1: C=AVG(B: A=1)

- 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	R1-R2-W2-W1
Dirty read	R1-W1-R2-abort1-W2
Nonrepeatable read	R1-R2-W2-R1
Ghost update	R1-R1-R2-R2-W2-W2-R1
Phantom insert	R1-W2 (new data)-R1

Schedule

- Sequence of input/output operations performed by concurrent transactions

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

$r_1, w_1 \in T_1$

$r_2, w_2 \in T_2$

Principles of Concurrency Control

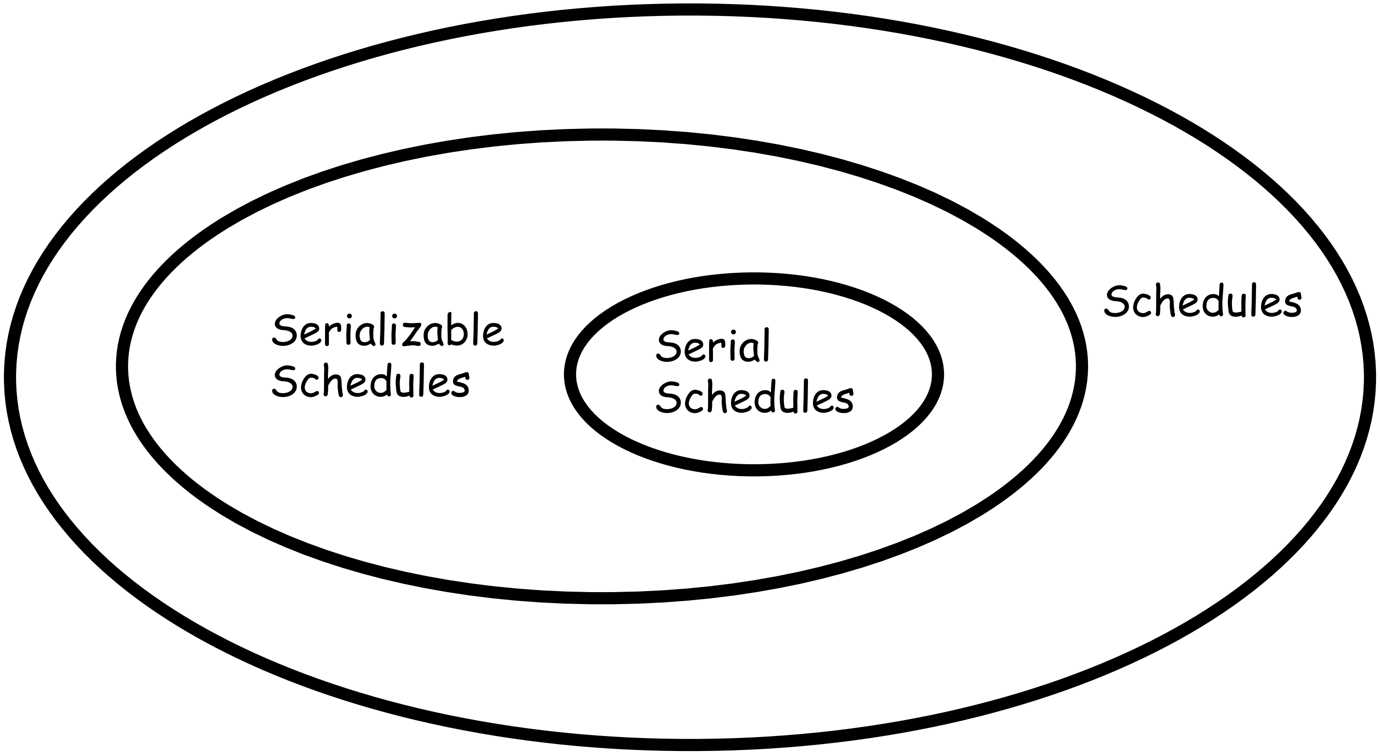
- Goal: to reject schedules that cause anomalies
- *Scheduler*: component that accepts or rejects the operations requested by the transactions
- *Serial schedule*: the actions of each transaction occur in contiguous sequences

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

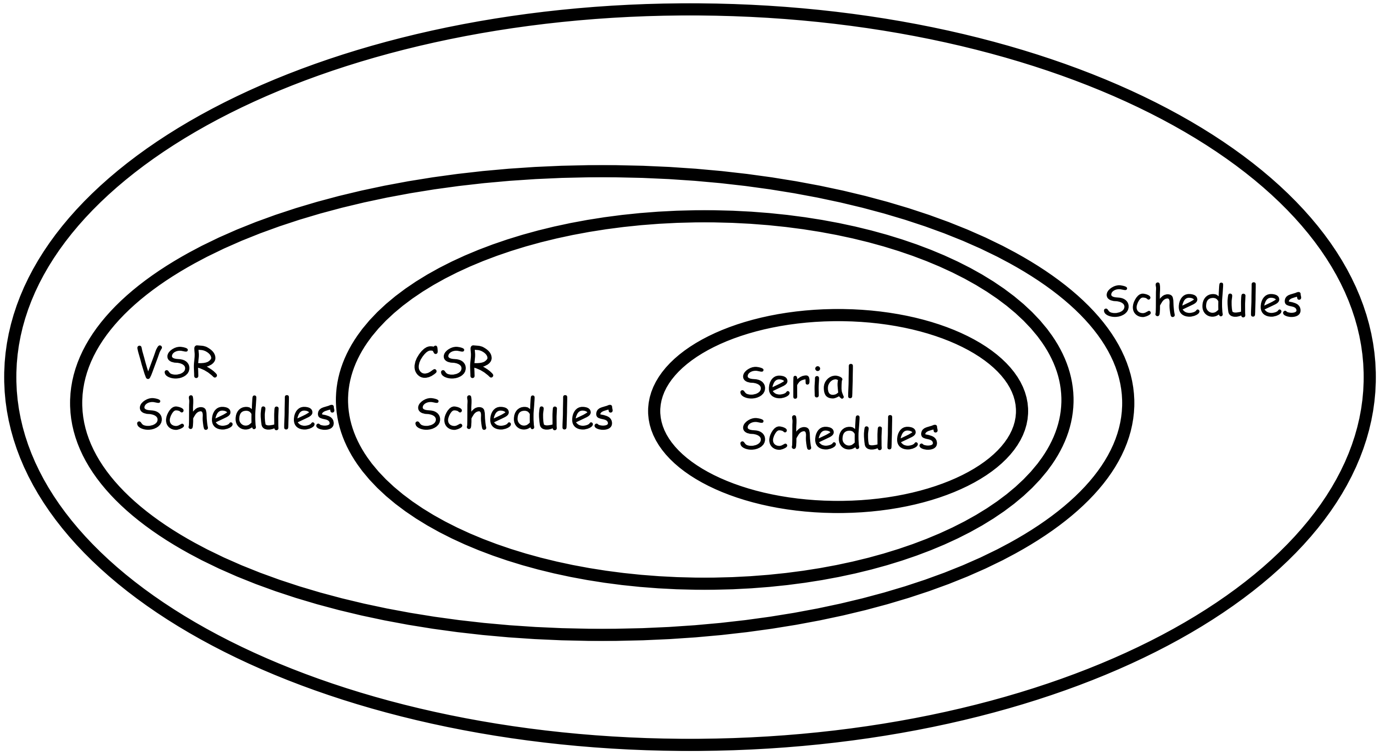
Principles of Concurrency Control

- *Serializable schedule*
  - produces the same results as some serial schedule on the same transactions
  - requires a notion of schedule equivalence
- Note: the class of acceptable schedules produced by a scheduler depends on the cost of equivalence checking
- Assumption
  - We assume that transactions are observed in the "past" (commit-projection) and we want to decide whether the corresponding schedule is correct
  - In practice, schedulers need to decide while the transaction is running

Basic Idea



CSR and VSR



View-serializability

- Preliminary definitions:
  - $r_i(x)$ *reads-from* $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ in S and there is no $w_k(x)$ between $r_i(x)$ and $w_j(x)$ in S
  - $w_i(x)$ 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 reads-from relations and the same final writes
- A schedule is *view-serializable* if it is equivalent to a serial schedule
- VSR is the set of view-serializable schedules

Examples of View-serializability

S_3 : $w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
S_4 : $w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
S_5 : $w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
S_6 : $w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$

- 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 2

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

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

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

- S_7 corresponds to a lost update
- S_8 corresponds to a non-repeatable read
- S_9 corresponds to a ghost update
- They are all not view-serializable

More Complex Example

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

S_{11}: $w_0(x), w_0(z), w_0(y), r_1(x), r_1(z), \underline{w_1(x),} r_2(x), w_2(y), r_3(z), w_3(z), w_3(y)$

- The serial order T0, T1, T2, T3 is not view equivalent to the above schedule.
- Let's try T0, T2, T1, T3

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's OK: $r_1(x)$ da $w_0(x)$,
$r_1(z)$ da $w_0(z)$,
$r_2(x)$ da $w_0(x)$,
$r_3(z)$ da $w_0(z)$,

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

It is VSR

Complexity of View-serializability

- Deciding view-equivalence of two given schedules can be done in polynomial time
- Deciding view-serializability of a generic schedule is an NP-complete problem

Conflict-serializability

- Preliminary definition:
  - Action a_i is conflicting with a_j ($i \neq j$) if both are operations on a common data item and at least one of them is a write operation.
    - *read-write* conflicts (*rw* or *wr*)
    - *write-write* conflicts (*ww*)

Conflict-serializability

- Conflict-equivalent schedules ($S_i \approx_C S_j$): S_i and S_j contain the same operations and all conflicting operation pairs occur in the same order
- S is a conflict-serializable schedule if it is conflict-equivalent to a serial schedule
- CSR is the set of conflict-serializable schedules

CSR and VSR

- Every conflict-serializable schedule is also view-serializable, but the converse is not necessarily true
- Counter-example:

  $r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$

  - View-serializable: view-equivalent to

    $r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$

  - Not conflict-serializable, due to the presence of:

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

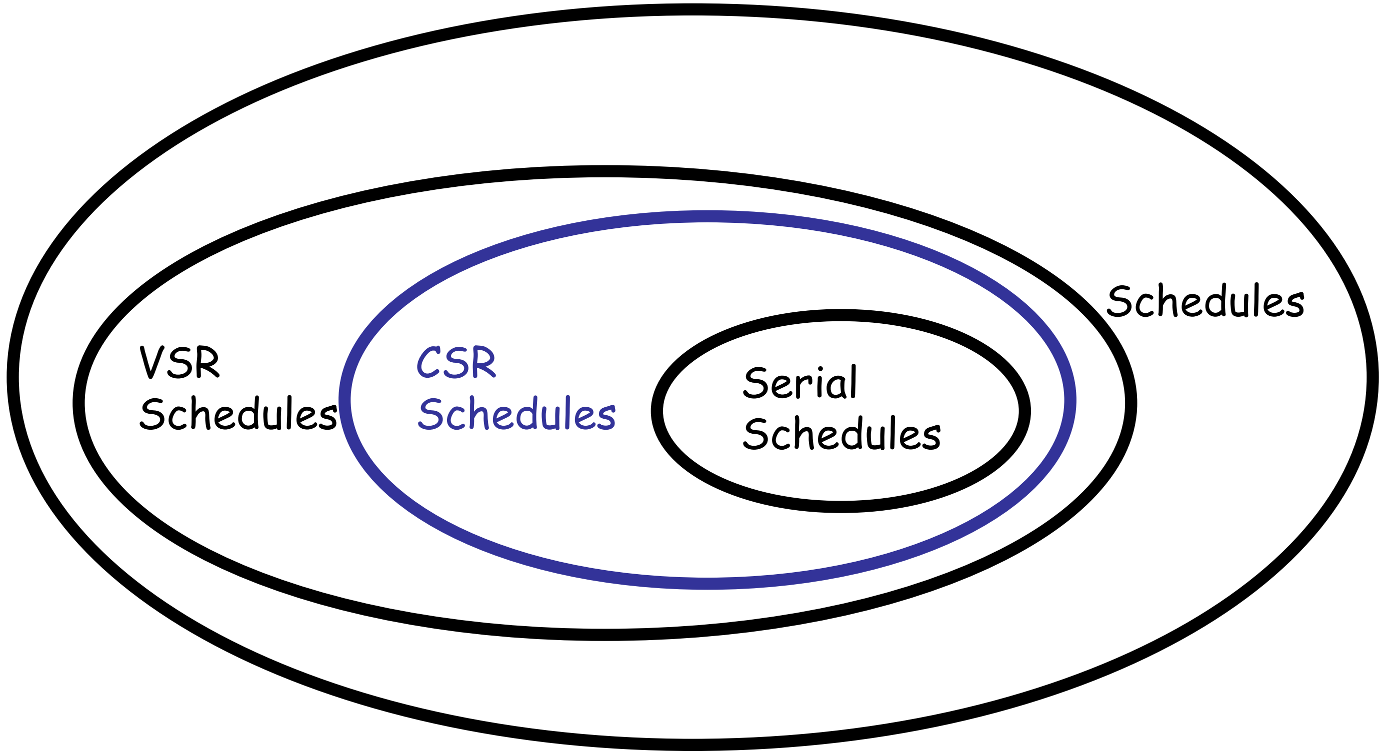
CSR implies VSR

- In order to prove that CSR implies VSR it suffices to prove that
  - Conflict-equivalence $\approx_C$ implies view-equivalence $\approx_V$,
  - i.e., if two schedules are $\approx_C$ then they also are $\approx_V$

CSR implies VSR

- Let's suppose $S_1 \approx_C S_2$. We prove that $S_1 \approx_V S_2$. These schedules have:
  - The same final writes: if they didn't, there would be at least two writes with 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 read-write pairs in a different order and therefore, as above, $\approx_C$ would be violated

CSR and VSR

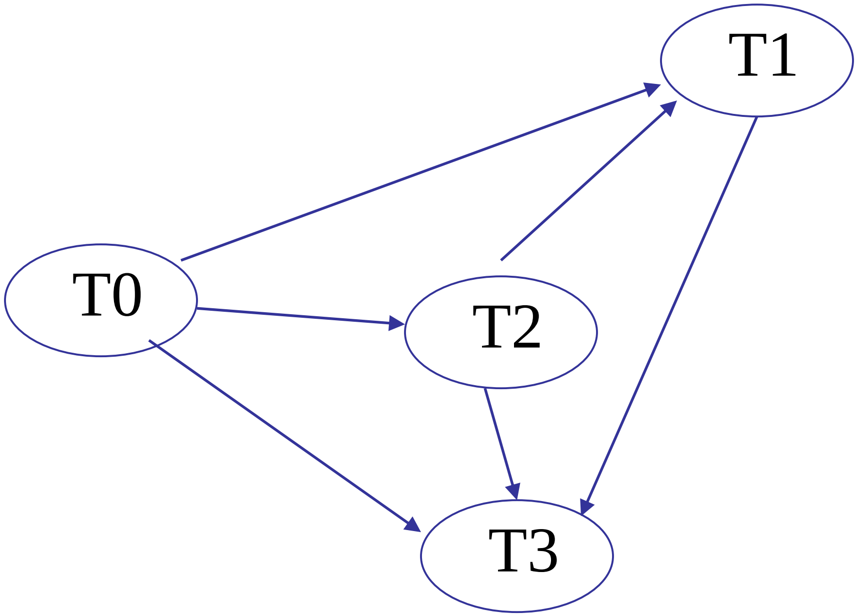


Testing Conflict-serializability

- 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 action a_i of T_i and an action a_j of T_j such that a_i precedes a_j
- Theorem:
  - A schedule is in CSR if and only if its conflict graph is acyclic

Example Test

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



CSR implies Acyclicity of the Conflict Graph

- Consider a schedule S in CSR. As such it is $\approx_C$ to a serial schedule.
- Suppose the order of transactions 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 S's 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

Properties of the Conflict Graph

- If S’s graph is acyclic then it has a *topological sort*, i.e., an ordering of the nodes such that the graph only contains arcs (i,j) with i<j
- The serial schedule whose transactions are ordered according to the topological sort is conflict-equivalent to S, because for all conflicting pairs (i,j) it is always i<j
    - In the example before: T0 < T2 < T1 < T3
    - In general there can be MANY topological sorts (i.e. 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 work “incrementally”, i.e., for each requested operation it should decide whether to execute it immediately or do something else
- It is not feasible to maintain the graph, update it and verify its acyclicity at each operation request

Locking

- It's the most common method in commercial systems
- A transaction is well-formed wrt 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`**
- When a transaction first reads and then writes an object it can:
  - Use a **`w_lock`**
  - Modify a **`r_lock`** into a **`w_lock`** (lock escalation)

Lock Primitives

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

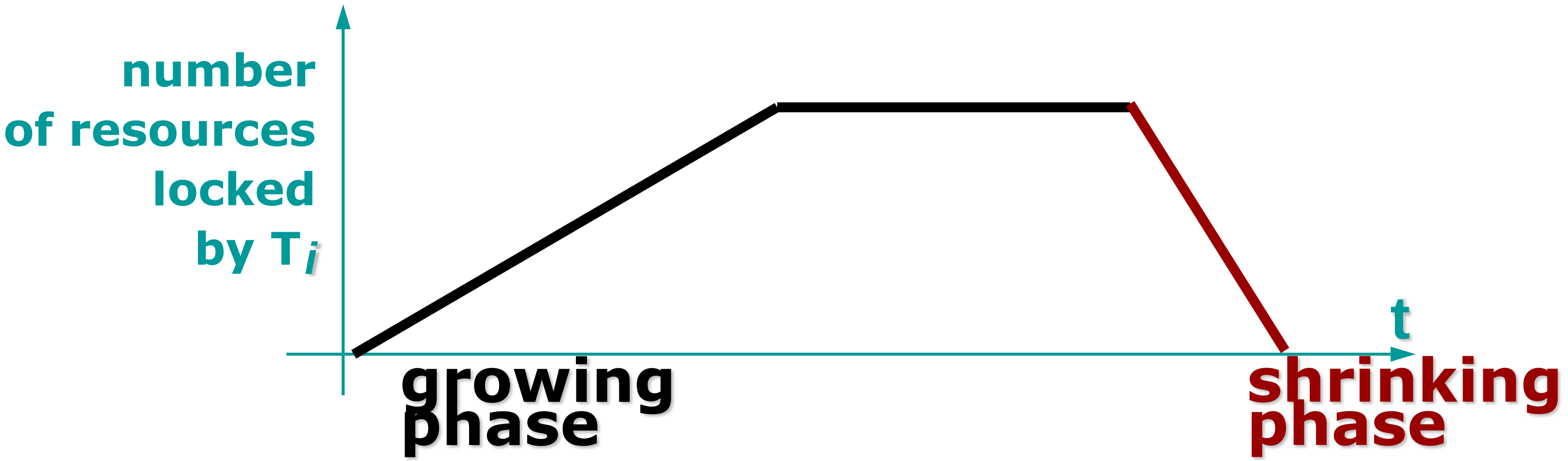
Behavior of the Lock Manager

- 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 STATE		
	FREE	R_LOCKED	W_LOCKED
`r_lock`	OK R_LOCKED	OK R_LOCKED	NO W_LOCKED
`w_lock`	OK W_LOCKED	NO R_LOCKED	NO W_LOCKED
`unlock`	ERROR	OK DEPENDS	OK FREE

Two-Phase Locking

- Requirements:
  - A transaction cannot acquire any other lock after releasing a lock



Serializability

- If a scheduler
  - uses well-formed transactions
  - grants locks according to conflicts
  - is two-phase
- Then it produces the schedule class called 2PL,

Schedules in 2PL are serializable

2PL and CSR

- 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

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

T1 releases T1 acquires

  - It is conflict-serializable

T3 < T1 < T2

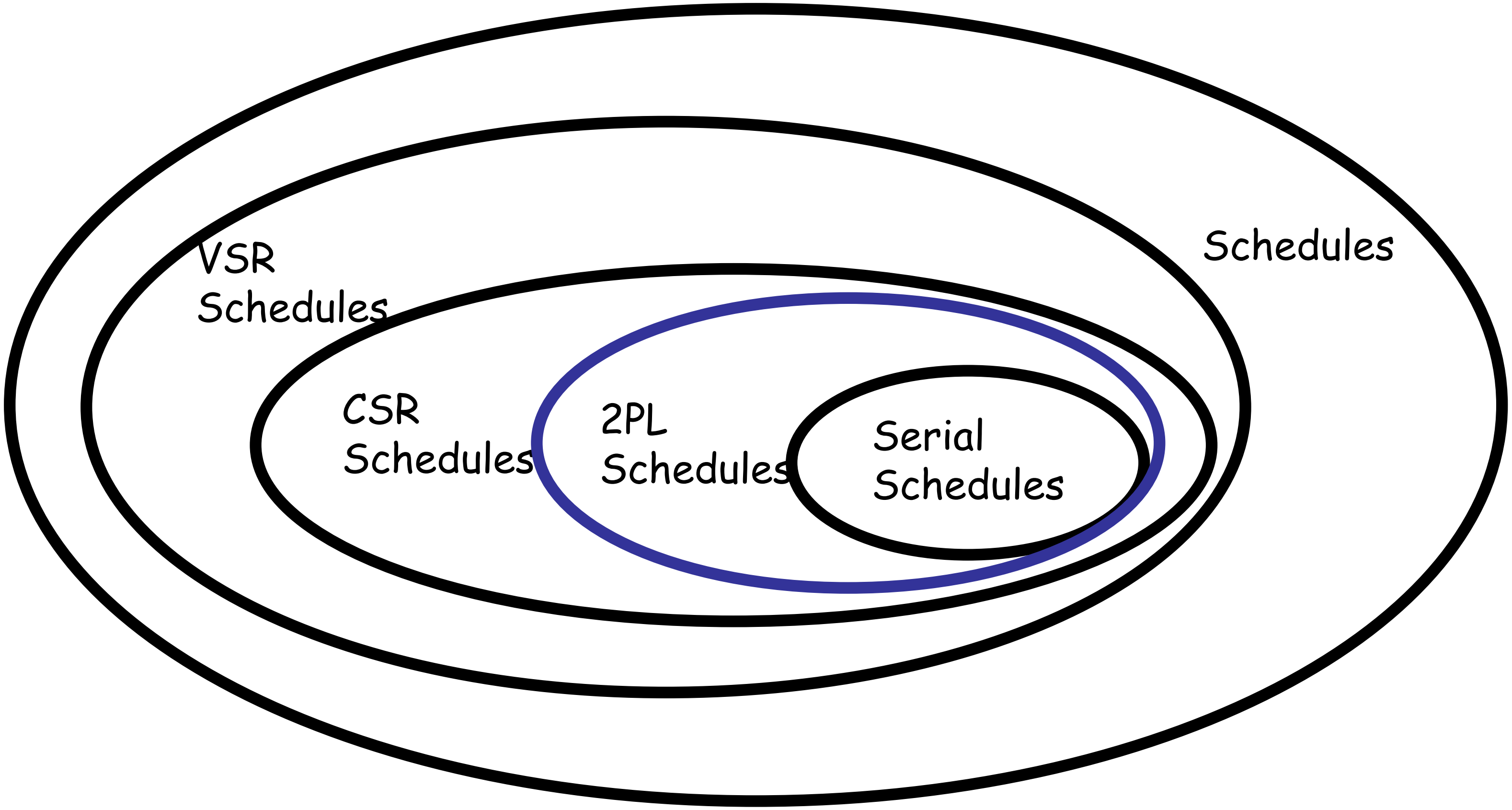
2PL implies CSR

- Consider for each transaction the moment in which it has all resources and is going to release the first one
- We sort the transactions by this temporal value and consider the corresponding serial schedule

2PL implies CSR

- We want to prove that this schedule is conflict-equivalent to S:
  - We then consider a conflict between an action from t_i and an action from the t_j's with $i<j$
  - Can they occur in the reverse order in S?
  - No, because then t_j should have released the resource in question before t_i has acquired it

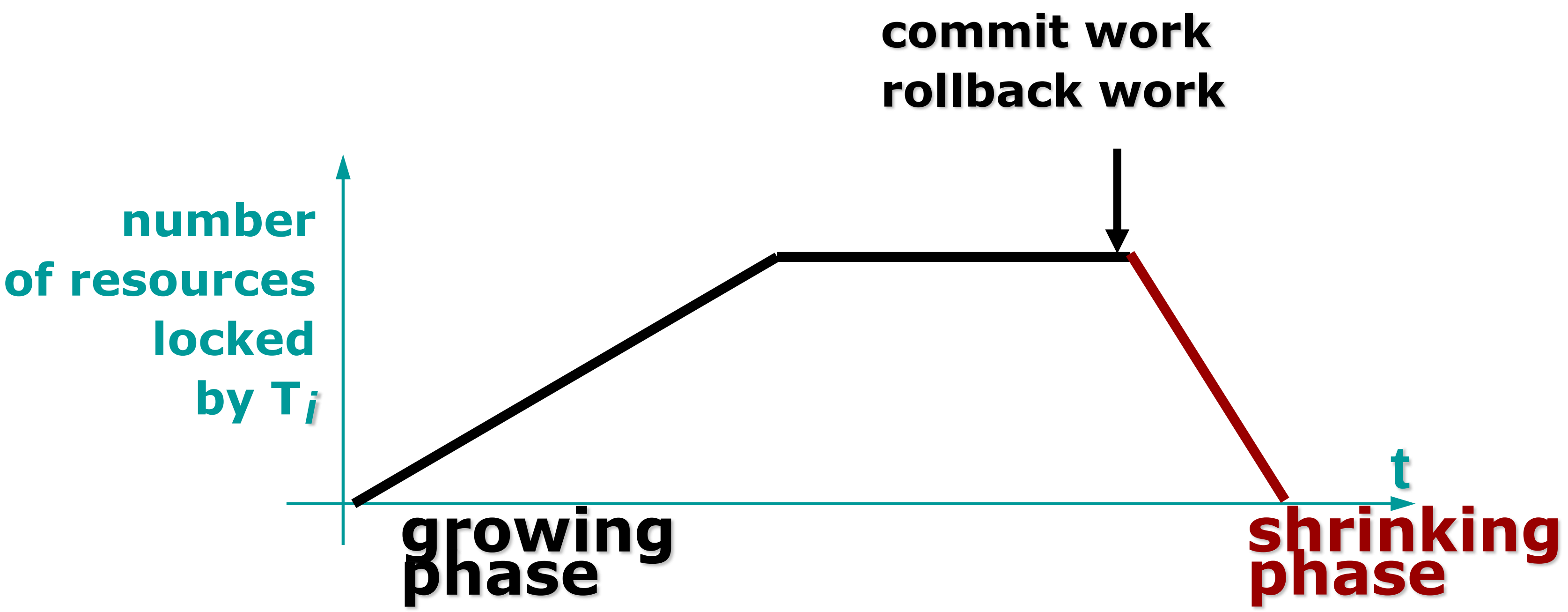
CSR, VSR and 2PL



Strict 2PL

- We were still using the hypothesis of commit-projection
- To remove this hypothesis, we need to add a constraint to 2PL, thus obtaining strict 2PL:
  - *Locks on a transaction can be released only after commit/rollback*
- This version of 2PL is used in commercial DBMSs

Strict 2PL in Practice



Implementation of 2-Phase Locking

- Lock tables are in reality **main memory data structures**.
  - Resource state is either *Free*, or *Read-Locked*, or *Write-locked*
  - To keep track of readers, every resource has also a "read counter"
  - Some late-ninety systems only supported exclusive locks (means: binary info for resources, no counter)
- A transaction asking for a lock is either granted a lock or **queued and suspended**, the queue is first-in first-out; there is a danger of:
  - Deadlock: endless wait
  - Starvation: individual transaction waiting forever
  - Starvation can occur for write transactions waiting for resources which are higly used for reading (e.g. index roots).

Isolation Levels in SQL:1999 (and JDBC)

- Writes are always applied strict 2PL (so update loss is avoided)
- `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

Predicate Locks

- With R(A,B), let: T=update R set B=1 where A=1
- Then, the lock is on predicate A=1
  - Cannot insert, delete, update any tuple satisfying such predicate
- Worst case:
  - On the entire relation
- If we are lucky:
  - On the index

Hierarchical Locking

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

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.

Resources Managed through Hierarchical Locking

lock

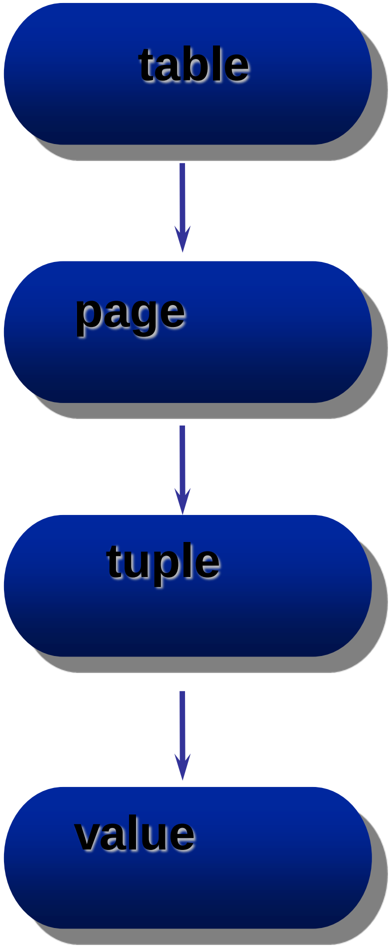
at the level of:

table

page

tuple

Attribute value



Reduced Granularity

Increased concurrency

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 define “intention of locking at lower levels of granularity”.
  - ISL: Intention of locking in shared mode
  - IXL: Intention of locking in exclusive mode
  - SIXL: Lock in shared mode with intention of locking in exclusive mode (SL+IXL)

Conflicts in Hierarchical Locks

Resource state

Request

	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 then going up in the hierarchy
- To request an SL or ISL lock on a non-root node, a transaction must hold an ISL or IXL lock on its parent node
- To request an IXL, XL or SIXL lock on a non-root note, a transaction must hold a SIXL or IXL lock on its parent node

Example

t1	t5
t2	t6
t3	t7
t4	t8

Page 1: t1,t2,t3,t4
Page 2: t5,t6,t7,t8

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

t1	t5
t2	t6
t3	t7
t4	t8

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

They are NOT in conflict!

Lock Sequences

t1	t5
t2	t6
t3	t7
t4	t8

t1	t5
t2	t6
t3	t7
t4	t8

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)

They are NOT in conflict!

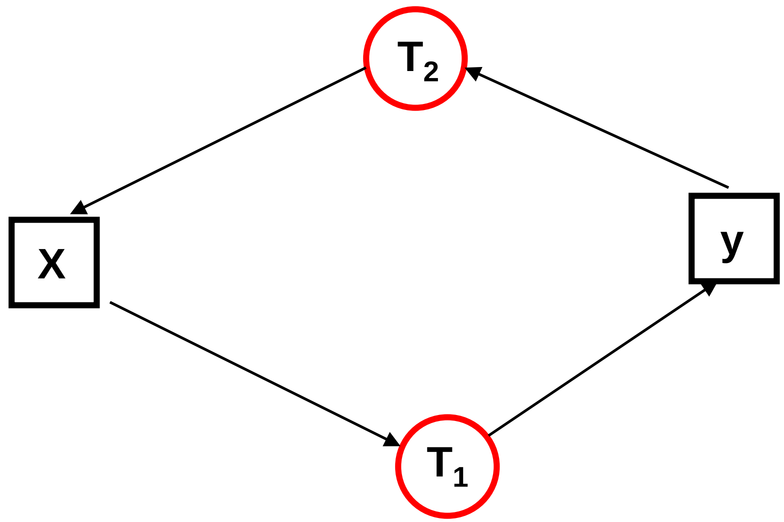
Deadlock

- Occurs because concurrent transactions hold and, in turn, require 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 the resources



Deadlock Resolution Techniques

- Timeout
  - Transactions killed after a long wait
- Deadlock prevention
  - Transactions killed when they COULD BE in deadlock
- Deadlock detection
  - Transactions killed when they ARE in deadlock

Timeout Method

- A transaction is killed after given waiting, assuming it is involved in a deadlock
- Simplest, most used method
- Timeout value is system-determined (sometimes it can be altered by the database administrator)
- The problem is choosing a proper value
  - Too long: useless wait
  - Too short: unrequired kills

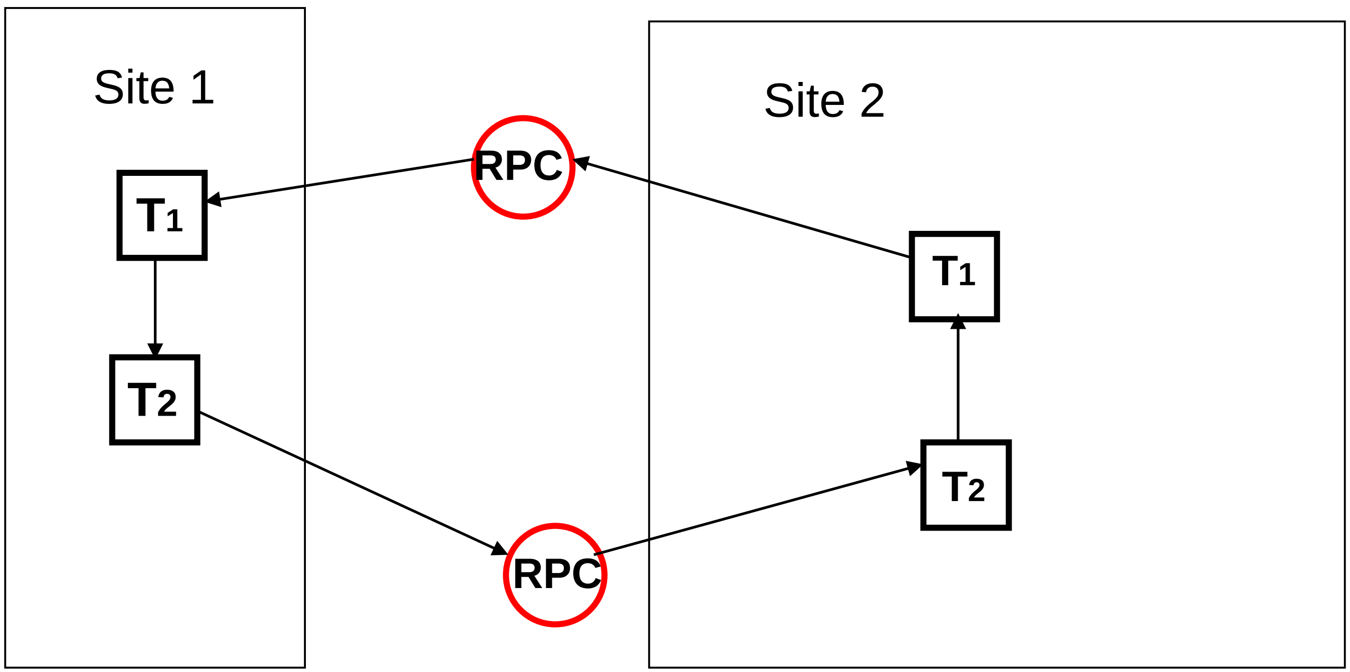
Deadlock Prevention

- Kills transactions that could cause cycles
- One possible scheme is:
  - assigning transaction numbers (assigning transactions an “age”)
  - killing transactions when “older” transactions wait for “younger” transactions
- Options for choosing the transaction to kill
  - Pre-emptive (killing the waiting transaction)
  - Non-pre-emptive (killing the requesting transaction)
- The problem: too many “killings” (waiting probability vs deadlock probability)

Deadlock Detection

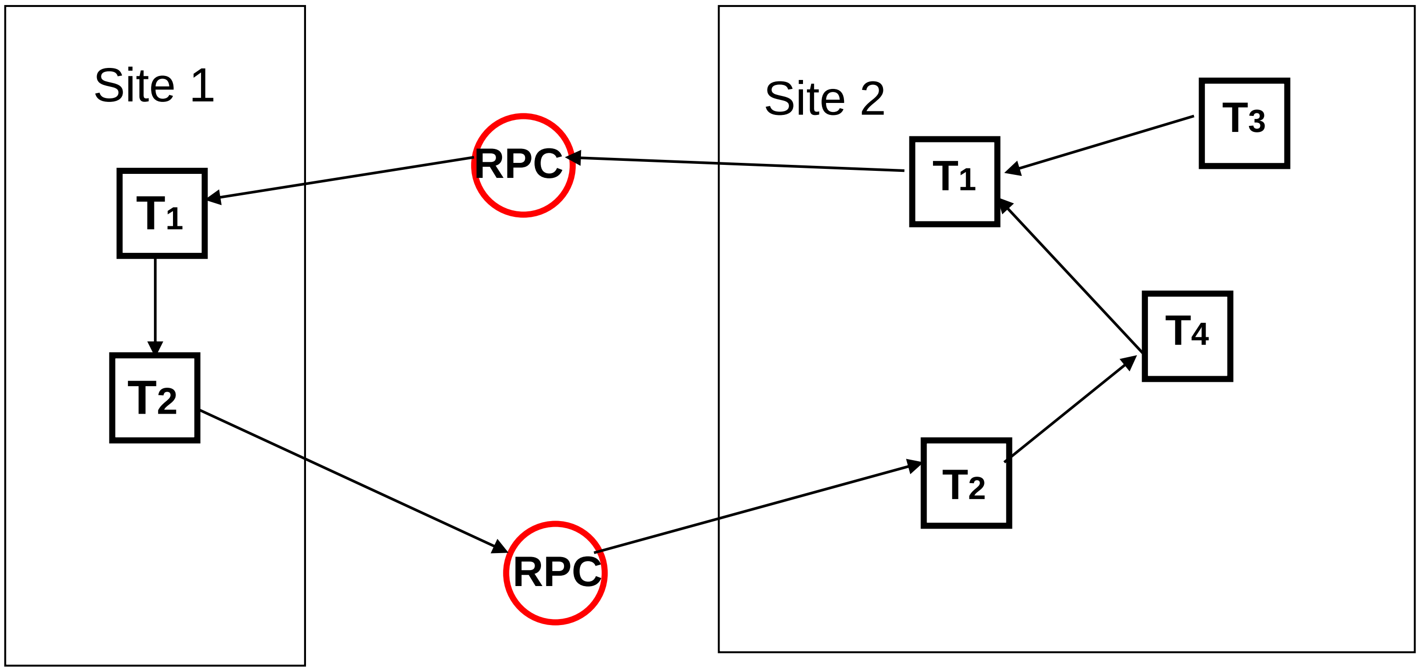
- Requires an algorithm for finding real cycles in the wait-for graph
  - Must work with distributed resources
  - Must be efficient and reliable
- The best solution: Obermark’s algorithm (DB2-IBM, published on ACM-Transactions on Database Systems)
  - Assumes synchronous transactions, each transaction works at a single site
  - Assumes communications via “remote procedure calls”
    - Both assumptions can be easily removed.

Distributed Deadlock Detection: Problem Setting



Potential Deadlock: at Site 1: E - T1 - T2 - E
at Site 2: E - T2 - T1 - E

Distributed Deadlock Detection: Problem Setting



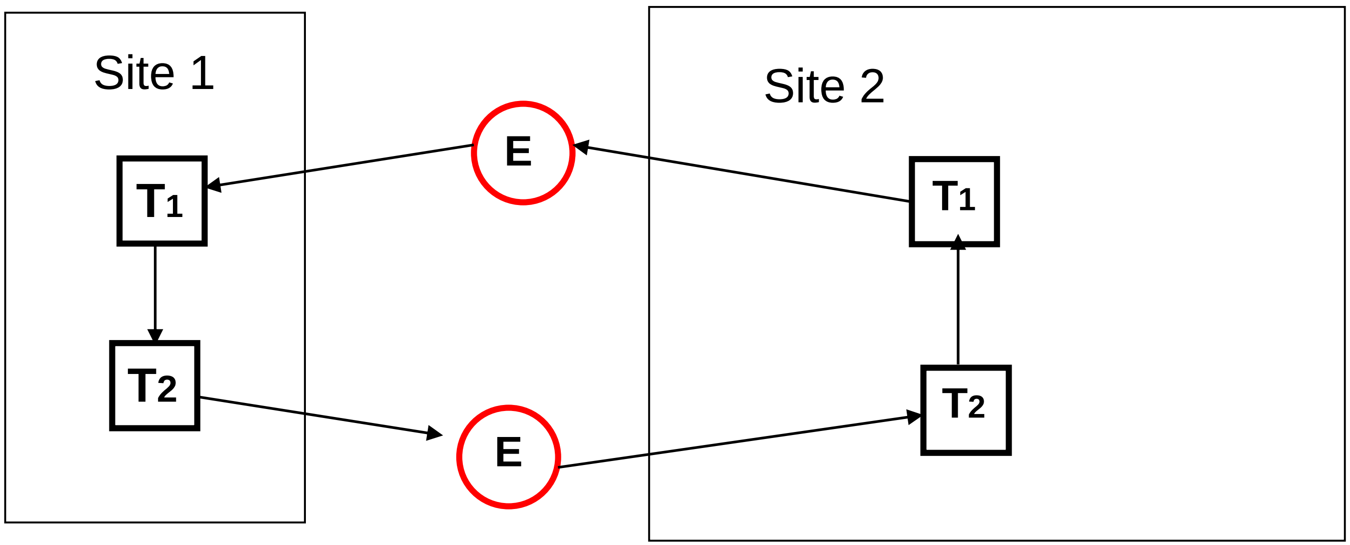
Potential Deadlock: at Site 1: E - T1 - T2 - E

at Site 2: E - T2 - T1 - E

Obermark's Algorithm

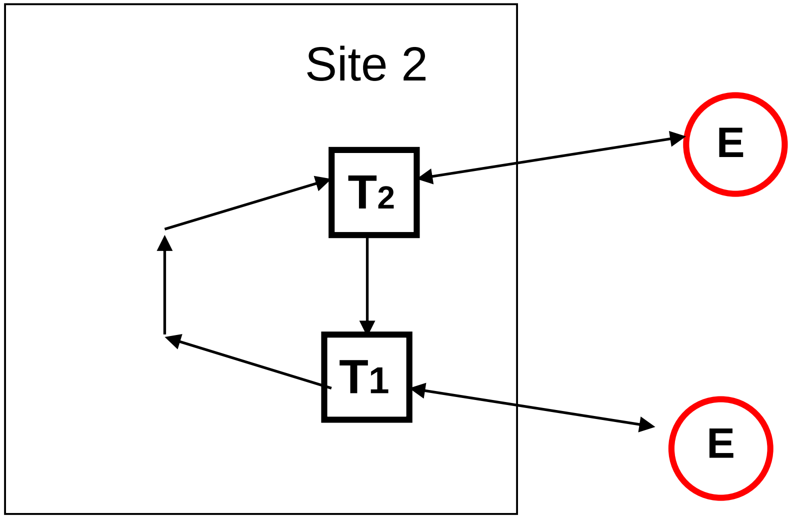
- Runs periodically at each site
- Consists of 4 steps
  - *Get potential deadlock from the "previous" nodes*
  - *Integrate deadlock cycles with local wait-for graph*
  - *Search deadlocks, if found kill one transaction*
  - *Build potential deadlock and transmit to the "next" node*
- Secondary objective: detect every cycle only once; achieved by define "previous" and "next" node as follows:
  - Potential deadlock transmitted along the RPC chain
  - Potential deadlock E -Ti-Tj-E transmitted only if i<J

Algorithm execution, 1



Potential Deadlock at Site 1: E - T1 - T2 – E sent to 2
at Site 2: E - T2 - T1 - E not sent to 1

Algorithm execution, 2



1. E - T1 - T2 – E sent to 2
2. at Site 2:
   E – T1 – T2 - E added
   Deadlock detected
   T1 or T2 killed (rollback)

Another example

- Initially: at site 1, E > T1 > T2 > E
  2, E > T2 > T3 > E
  3, E > T3 > T1 > E
- Sites 1 and 2 can send info, 3 cannot
- Start with 1 sending to 2
- At site 2, a new potential dedadlock E > T1 > T3 > E is found by combining the incoming E > T1 > T2 > E and present E > T2 > T3 > E; this is sent to 3.
- At site 3, E > T1 > T3 > E is combined with E > T3 > T1 > E , the deadlock is found, and either T1 or T3 is killed.

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 $o(1/n^2)$
- They do occur (once every minute for a mid-size bank)
- Probability rises linearly with the number of transactions and quadratically with transaction size (number of lock requests)

Update Lock

- The most frequent deadlock occurs when 2 concurrent transactions start by reading the same resource and then they decide to write and escalate or write-lock it.
- To avoid this situation, systems offer the UPDATE LOCK (UL) – used by transactions that may change the resource value.

Request	State		
	SL	UL	XL
SL	OK	OK	No
UL	OK	No	No
XL	No	No	No

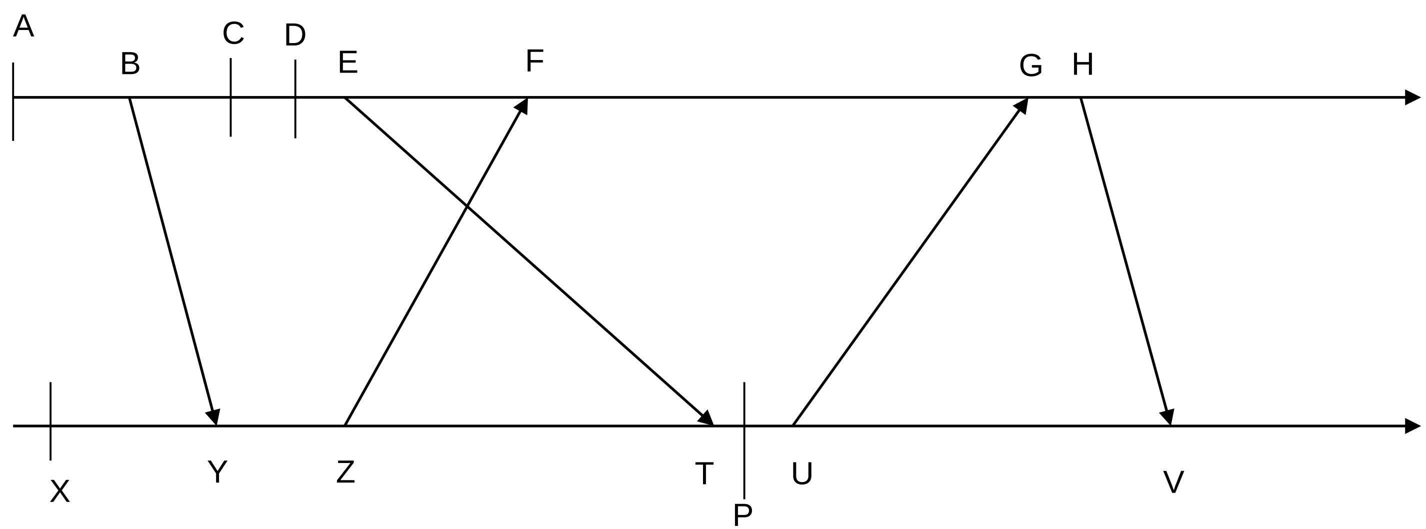
Concurrency Control Based on Timestamps

- Alternative to 2PL
- Timestamp:
  - *Identifier defining 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 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 beyond the timestamp of the send.

Example of timestamp assignment



A(1,1), B(2,1), X(1,2), Y(2,2), C(3,1), D(4,1), E(5,1), Z(3,2), F(6,1), T(5,2), P(6,2), U(7,2), G(8,1), H(9,1), V(9,2)

Timestamp Mechanism

- The scheduler has two counters: RTM(*x*) and WTM(*x*) for each object
- The scheduler receives read and write requests with timestamps:
  - *read(x,ts)*:
    - If *ts* < WTM(*x*) the request is rejected and the transaction killed
    - Else, the request is granted and RTM(*x*) is set to max(RTM(*x*), *ts*)
  - *write(x,ts)*:
    - If *ts* < WTM(*x*) or *ts* < RTM(*x*) the request is rejected and the transaction killed
    - Else, the request 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

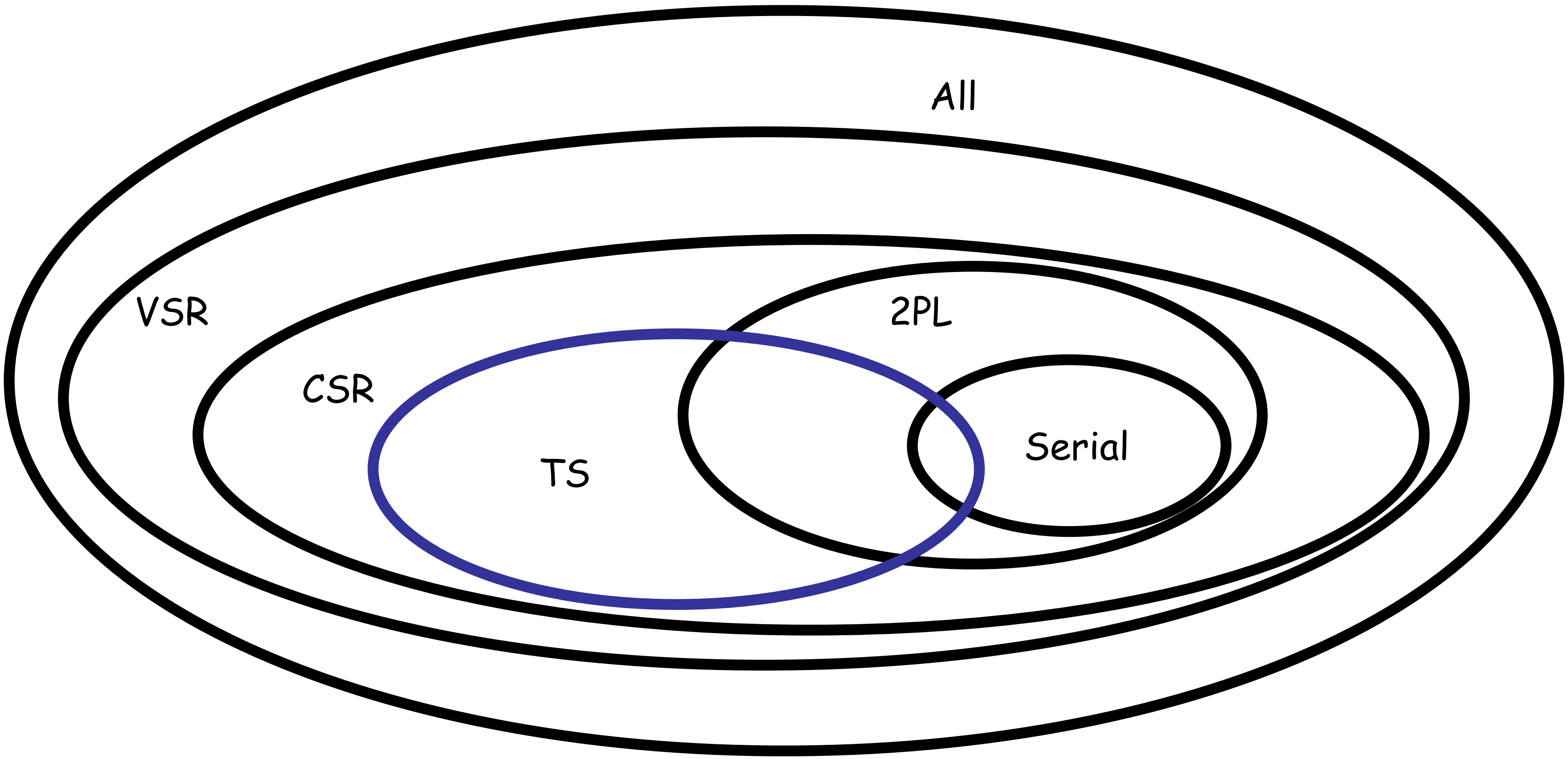
- They are incomparable
  - Schedule in TS but not in 2PL

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



2PL vs. TS

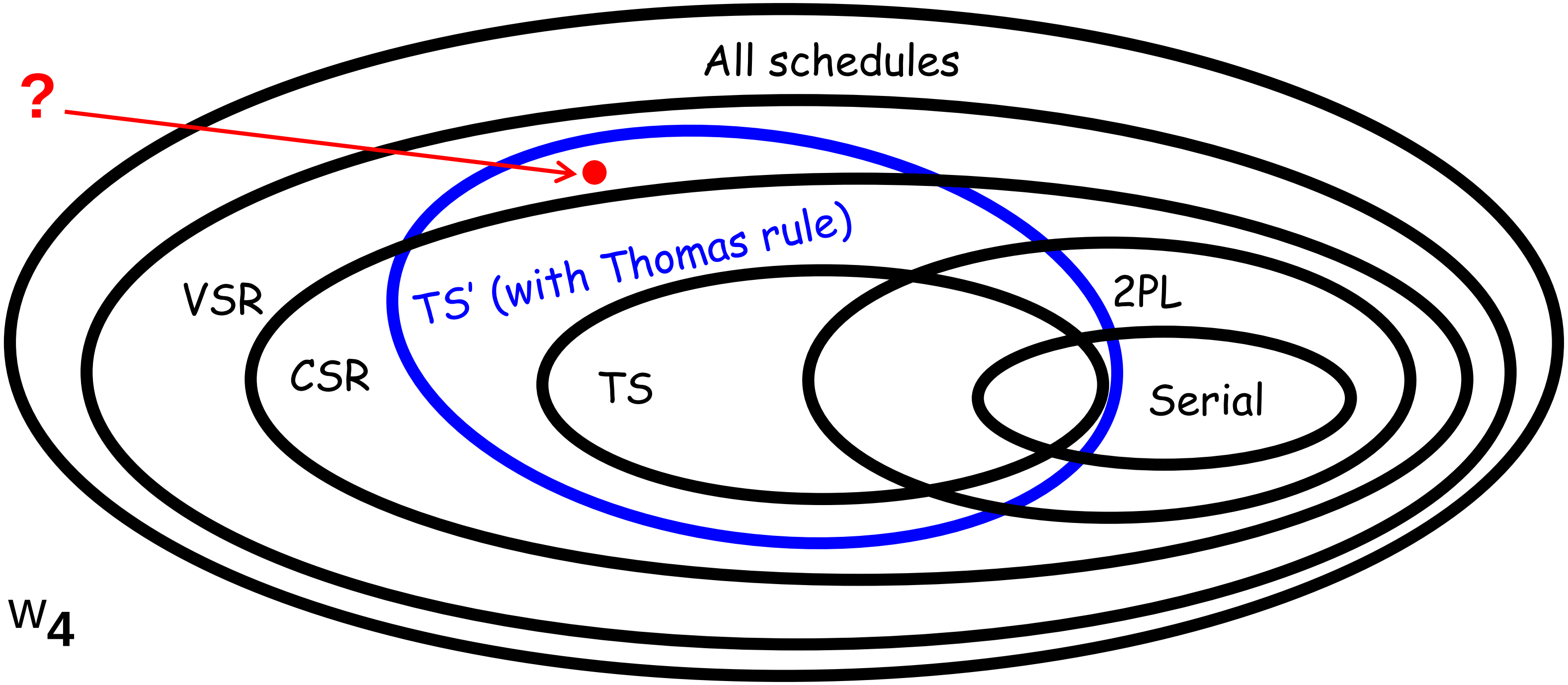
- In *2PL* transactions can be waiting. In *TS* they are killed and restarted
- The serialization order with *2PL* is imposed by conflicts, while in *TS* it is imposed by the timestamps
- The necessity of waiting for commit of transactions causes long delays in strict *2PL*
- *2PL* can cause deadlocks (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, the request 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, the request 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 1	*RTM(x) = 10*
read(x,12)	ok on 2	*RTM(x) = 12*
write(x,13)	ok	$N=3$, $WTM(x_3) = 13$

Snapshot isolation

- 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

T1: update Balls set Color=White where Color=Black

T2: update Balls set Color=Black where Color=White

- A serializable execution of T1 and T2 would produce at the end a configuration with balls that are either all white or all black
- An SI execution may just swap the colors