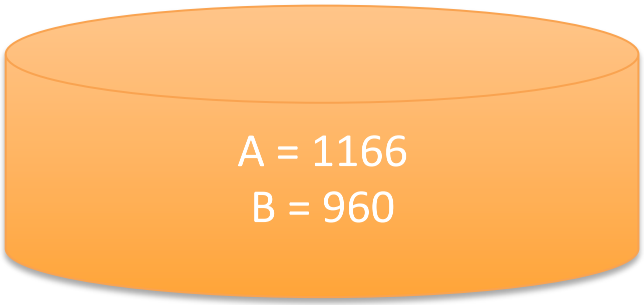
Concurrency

Carsten Schürmann

Abstract away from Details

T1	T2
BEGIN	
A = A + 100	
	BEGIN
	A = A * 1.06
	B = B * 1.06
	COMMIT
B = B - 100	
COMMIT	



T1	T2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Interleaved Execution Anomalies

- **Read-Write** conflicts (R-W)
- **Write-Read** conflicts (W-R)
- **Write-Write** conflicts (W-W)

Definition: Two operations conflict if

- They are part of two different transactions
- They are on the same object
- At least one of the operations is a write.

Serial Schedule

T1	T2
BEGIN	
A = A + 100	
	BEGIN
	A = A * 1.06
B = B - 100	
COMMIT	
	B = B * 1.06
	COMMIT

≡

T1	T2
BEGIN	
A = A + 100	
B = B - 100	
COMMIT	
	BEGIN
	A = A * 1.06
	B = B * 1.06
	COMMIT

Some More Mathematics

Last time: Equivalent schedules

Definition: Two schedules conflict equivalent iff

- They contain the same (respective) actions
- Every pair of conflicting actions is ordered the same way

Definition: A schedule is conflict serializable iff it is conflict equivalent to some serial schedule.

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

$\equiv$

T1	T2
BEGIN	
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

Dependency Graph

Definition:

Nodes: Names of transactions

Edges:

An operation O_1 of T_1 conflicts with an operation O_2 of T_2 and O_1 appears before O_2.

Alias: Precedence Graph

Theorem

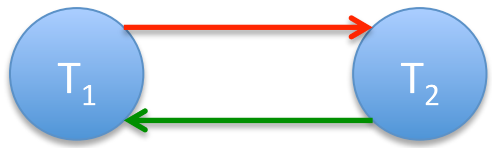
A schedule is conflict serializable if and only if its dependency graph is acyclic.

Example 1:

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Example 1:

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	



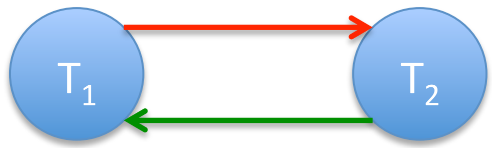
Dependency found:
The output of T_1 depends on T_2 and vice versa.

Example 2: Lost Update

T1	T2
BEGIN	BEGIN
R(A)	
A = A - 1	
	R(A)
	A = A - 1
	W(A)
	COMMIT
W(A)	
COMMIT	

Example 2: Lost Update

T1	T2
BEGIN	BEGIN
R(A)	
A = A - 1	
	R(A)
	A = A - 1
	W(A)
	COMMIT
W(A)	
COMMIT	



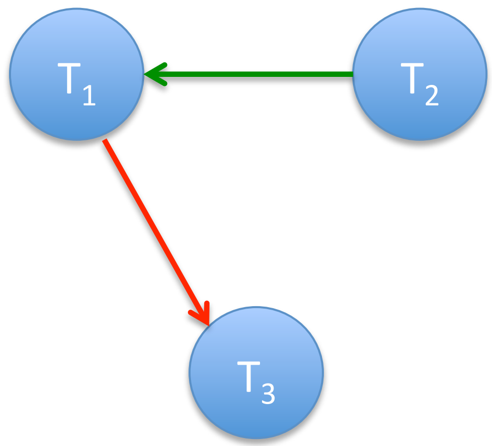
Dependency found:
The output of T_1 depends on T_2 and vice versa.

Example 3: Threesome

T1	T2	T3
BEGIN		
R(A)		
W(A)		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
	COMMIT	
R(B)		
W(B)		
COMMIT		

Example 3: Threesome

T1	T2	T3
BEGIN		
R(A)		
W(A)		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
	COMMIT	
R(B)		
W(B)		
COMMIT		



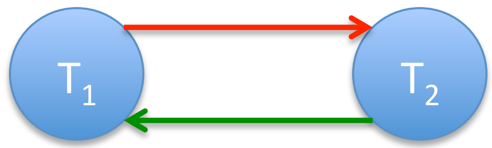
No dependency found:
A serial schedule exists
Order: T_2, T_1, T_3

Example 4: Lost Update

T1	T2
BEGIN	BEGIN
R(A)	
A = A - 1	
W(A)	
	R(A)
	Sum = A
	R(B)
	Sum = Sum + B
	PRINT (Sum)
R(B)	COMMIT
B = B +1	
W(B)	
COMMIT	

Example 4: Lost Update

T1	T2
BEGIN	BEGIN
R(A)	
A = A - 1	
W(A)	
	R(A)
	Sum = A
	R(B)
	Sum = Sum + B
	PRINT (Sum)
R(B)	COMMIT
B = B +1	
W(B)	
COMMIT	



Dependency found:
The output of T_1 depends on T_2 and vice versa.

View Serializability

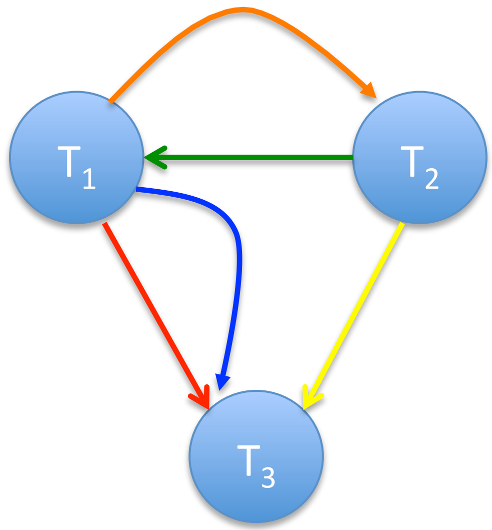
Two schedules S_1 and S_2 are view-equivalent

($S_1 \equiv_S S_2$) iff

- If T_1 reads initial value of A in S_1, then T_1 also reads initial value of A in S_2
- If T_1 reads value of A written by T_2 in S_1, then T_1 also reads value of A written by T_2 in S_2
- If T_1 reads final value of A in S_1, then T_1 also reads final value of A in S_2

Example

T1	T2	T3
BEGIN		
R(A)		
	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT



Dependency found: This schedule is not conflict serializable, but …

Example

T1	T2	T3
BEGIN		
R(A)		
	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

$\equiv_S$

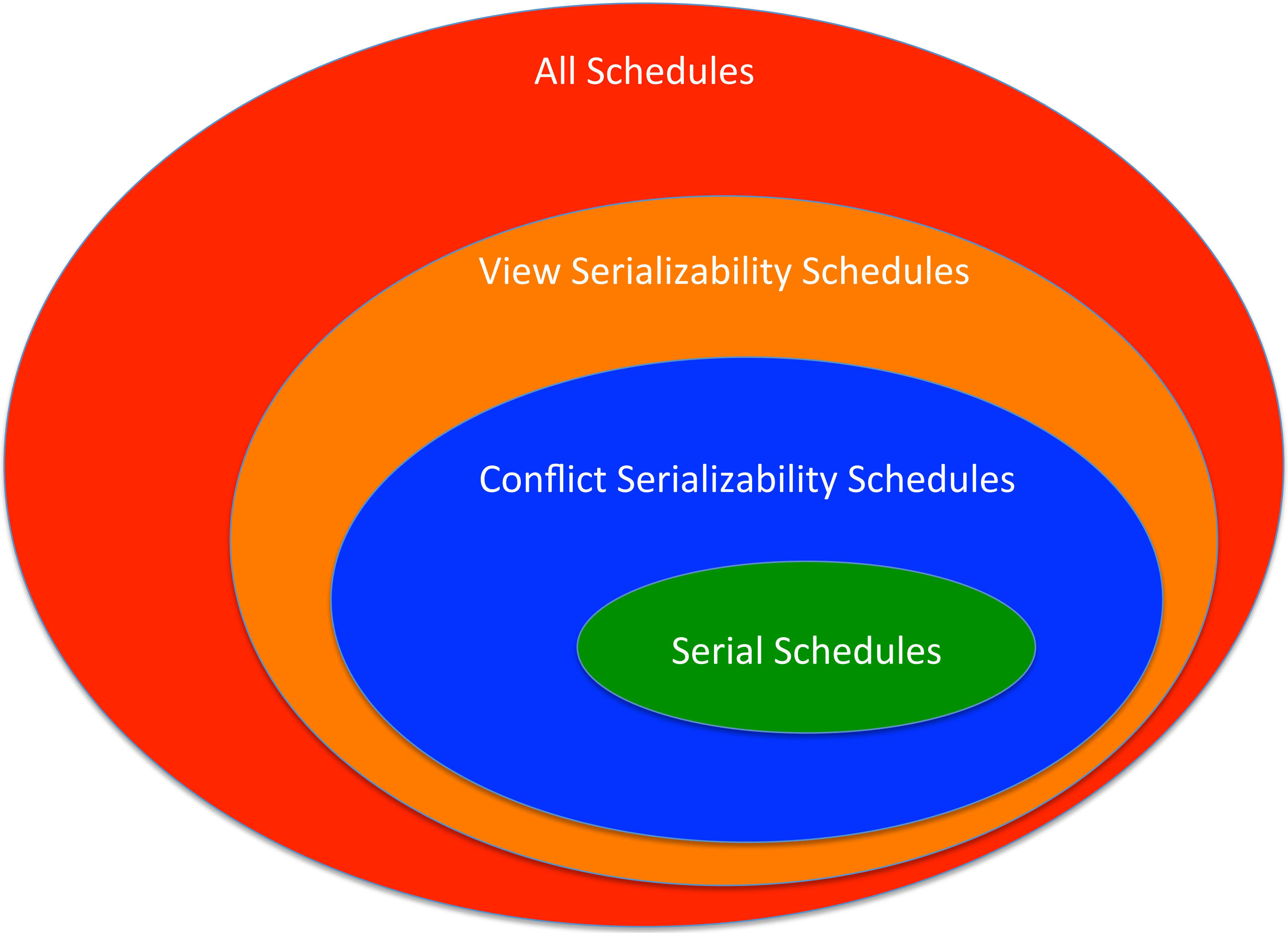
T1	T2	T3
BEGIN		
R(A)		
W(A)		
COMMIT	BEGIN	
	W(A)	
	COMMIT	BEGIN
		W(A)
		COMMIT

Serializability

View Serializability

- allows (slightly) more schedules than Conflict Serializability does
- difficult to enforce efficiently

In practice Conflict Serializability is used.


All Schedules
View Serializability Schedules
Conflict Serializability Schedules
Serial Schedules

Deadlocks

Oh no!

T1	T2
BEGIN	BEGIN
X-LOCK(A)	
A = A - 1	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

Lock Manager
Granted (T1 → A, X)
Granted (T2 → B, S)
Denied
Denied

Deadlocks

Two ways of dealing with deadlocks

- Deadlock prevention
- Deadlock detection

Common way of dealing with deadlocks:

Timeouts

Deadlock Detection

Definition: Waits-for graph

Nodes: Names of transactions

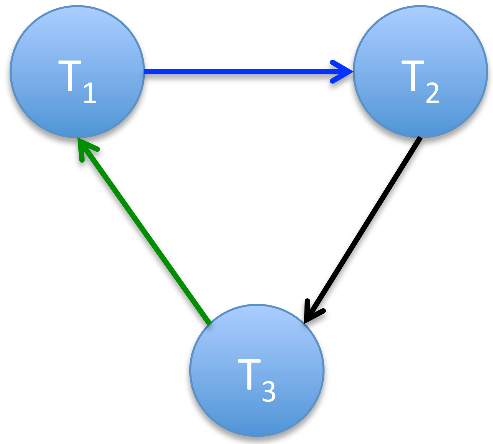
Edges:

T_1 is waiting for T_2 to release a lock.

Computed and checked periodically on the fly

Deadlock Detection

T1	T2	T3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
S-LOCK(D)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)



Corrective Measures

If there is a cycle:

- Choose a transaction
  - For example, by age, number of locks acquired…
- Roll-back
  - Minimally (Just until the locks are freed)
  - Completely
  - May cascade

Deadlock Prevention

- When a transaction asks for a lock that is held by another transaction kill one of the two
- No need to maintain a wait-for-graph
- No need for deadlock detection

Dead Lock Prevention

Record time stamp with transactions

Seniority means priority

Old waits for young

Wait-Die: If T_1 has higher priority than T_2, then T_1 waits for T_2 otherwise T_1 aborts.

Young waits for old

Wound-Wait: If T_1 has higher priority than T_2, then T_2 aborts otherwise T_1 waits for T_2.

Restarted transactions keep same time stamp

Example 1

T1	T2
BEGIN	
	BEGIN
	X-LOCK(A)
X-LOCK(A)	

Wait – Die

T_1 waits

Wound– Wait

T_2 aborted

Example 2

T1	T2
BEGIN	
X-LOCK(A)	
	BEGIN
	X-LOCK(A)

Wait – Die

T_2 aborted

Wound– Wait

T_2 wait

Discussion

Theorem: These schemes guarantee no deadlock.

Proof: Wait-for graphs has no loops!

T1	T2	T3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
S-LOCK(D)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Looking in practice

Manually locking usually not required…

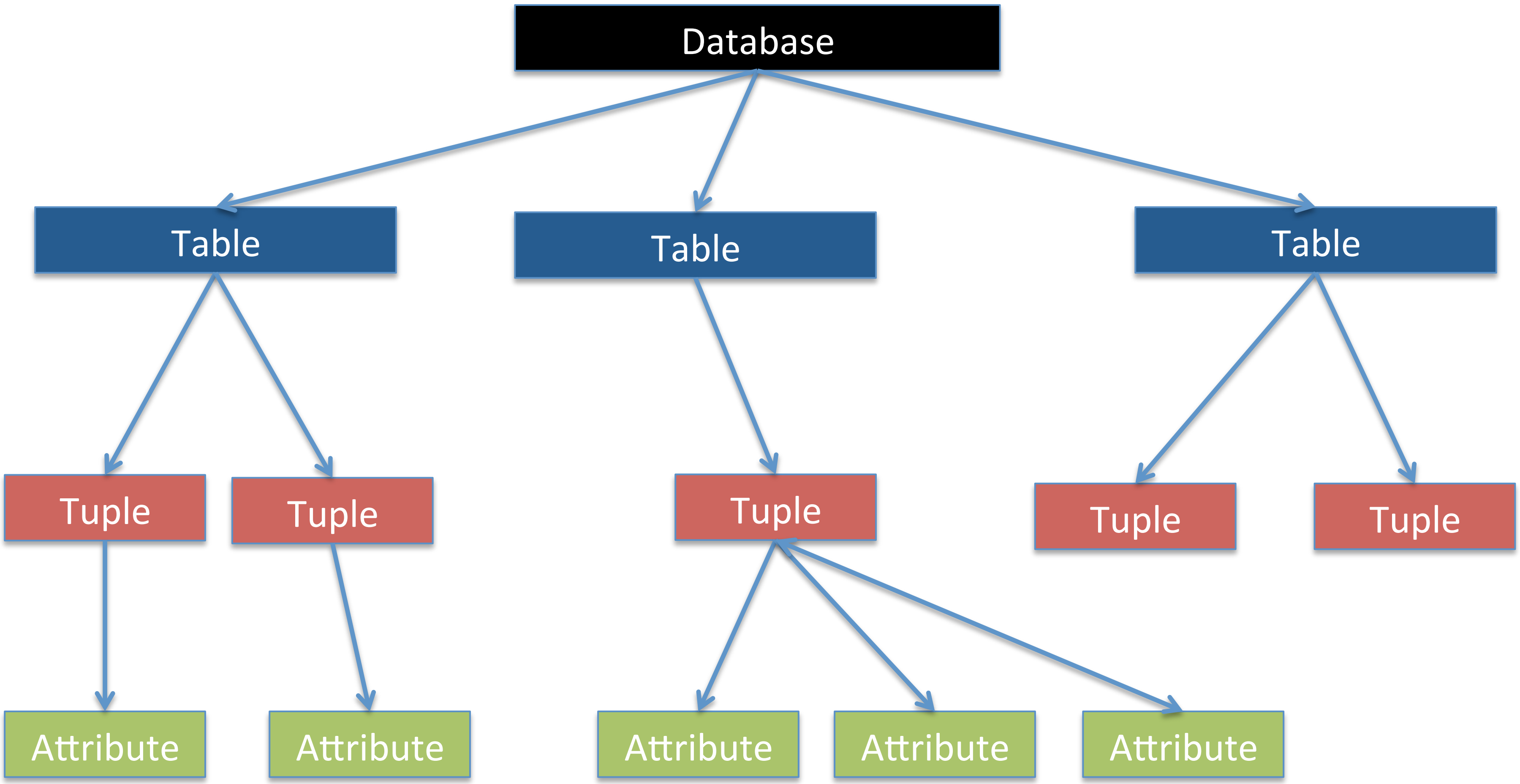
But you can

```
LOCK TABLE <table> READ;

LOCK TABLE <table> WRITE;
```

Lock Granularities

Database Lock Hierarchy

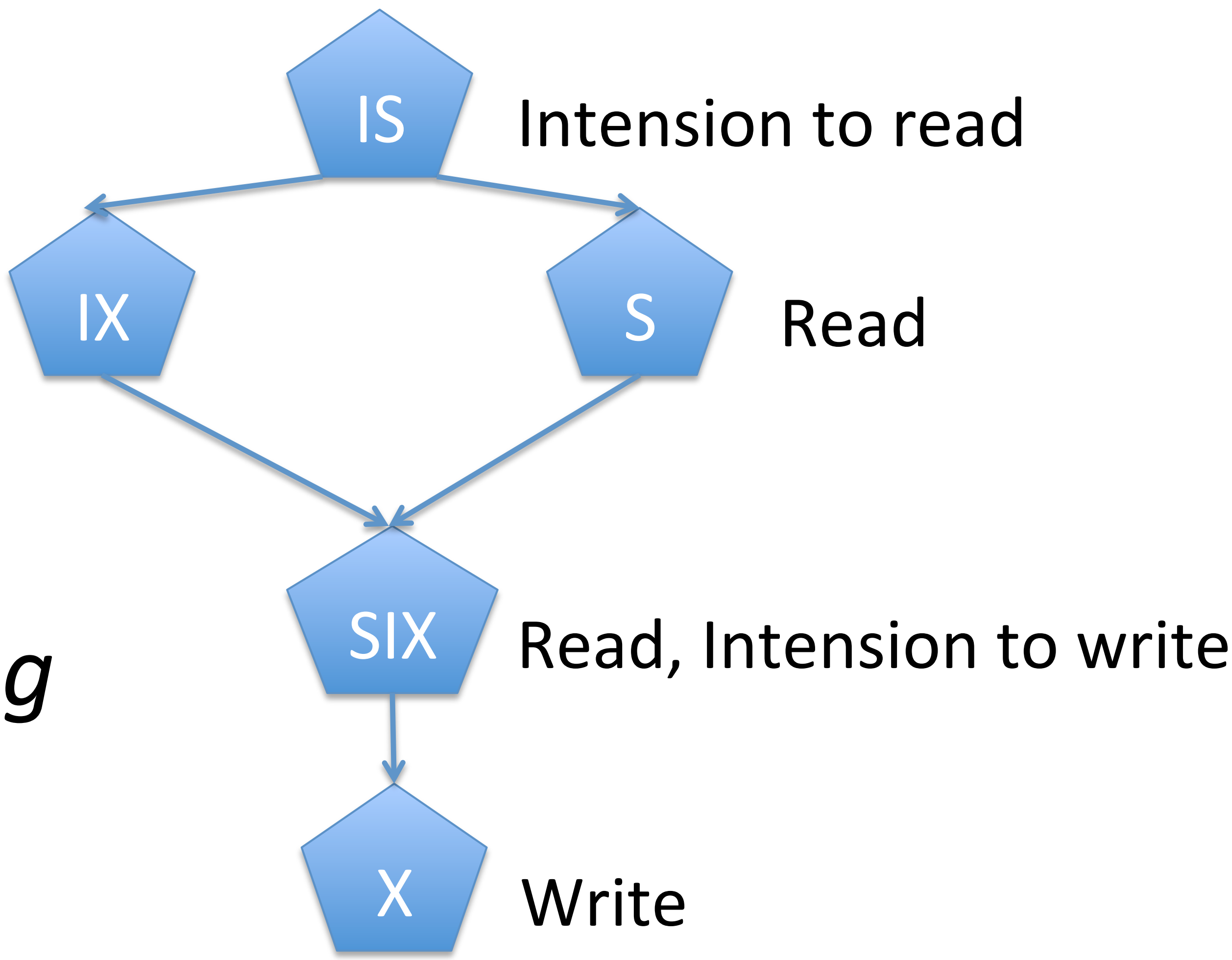


Example

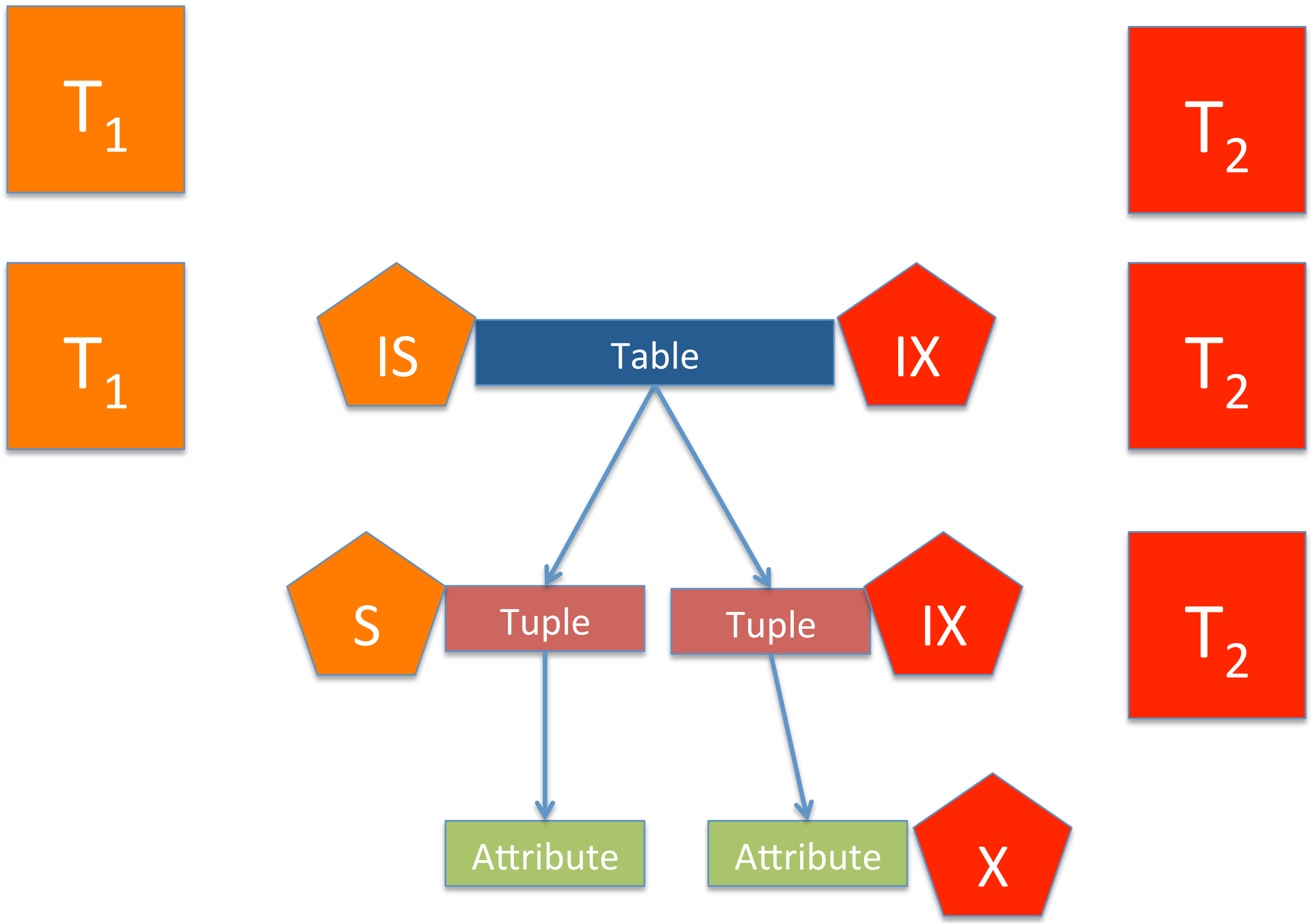
Transaction: Transfer 1000kr from Jesper's to Carsten's account.

What do we have to lock?

Intention Locks



Summarize what is going on below in the tree.



T_1
T_2
T_1
IS
Table
IX
T_2
S
Tuple
Tuple
IX
T_2
Attribute
Attribute
X

Intention Lock Compatibility

	IS	IX	S	SIX	X
IS	✔	✔	✔	✔	✘
IX	✔	✔	✘	✘	✘
S	✔	✘	✔	✘	✘
SIX	✔	✘	✘	✘	✘
X	✘	✘	✘	✘	✘

Locking Protocol

- Each transaction acquires the lock at the root
- To get S or IS locks, the transaction must hold at least IS on the parent node
- To get X, IX or SIX on a node, the transaction must hold at least IX on the parent node

The Phantom Problem

Phantom Problem

	T1	T2
	BEGIN	BEGIN
28	SELECT MAX(age) FROM students WHERE grade = 12	
		INSERT INTO students (cpr=15, age=35, grade =12)
35	SELECT MAX(age) FROM students WHERE grade = 12	
	COMMIT	COMMIT

Phantom Problem

We locked only existing records and not new ones.

Conflict Serializability works only, if the set of objects is fixed!

How to fix this:

Predicate Locking or Index Locking

Predicate Locking

Lock all records that satisfy a logical predicate

(locks automatically to those records just added)

Example:

Lock records, s.t. grade = 12

Discussion: Lots of overhead

Index Locking

Assume that there is a dense index on grade.

Lock index page that contains all tuples with grade = 12

Index locking is a special case of predicate locking, ust more efficient.

Transaction Isolation Levels

Weaker Levels of Consistency

- Serializability is useful because it allows programmers not to worry about concurrency issues.
- But enforcing it may allow too little concurrency and limit performance.
- We may want to use a weaker level of consistency to improve scalability.

Isolation Levels

- Controls the extent that a transaction is exposed to the actions of other concurrent transactions.
- Provides for greater concurrency at the cost of exposing transactions to uncommitted changes:
  - Dirty Reads
  - Unrepeatable Reads
  - Phantom Reads

Isolation Levels

SERIALIZABLE:

No phantoms, all reads repeatable, no dirty reads

REPEATABLE READS:

Phantoms may happen

READ COMMITTED:

Phantoms and unrepeatable reads may happen

READ UNCOMMITTED:

All of them may happen

Isolation Levels

	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Isolation Levels

SERIALIZABLE:

Obtain all locks first; plus index locks, plus strict 2PL.

REPEATABLE READS:

Same as above, but no index locks.

READ COMMITTED:

Same as above, but **S** locks are released immediately.

READ UNCOMMITTED:

Same as above, but allows dirty reads (no **S** locks).

SET TRANSACTION ISOLATION LEVEL <level>;

	Default	Maximum
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Aerospike	READ COMMITED	READ COMMITED
Greenplum 4.1	READ COMMITED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITED	READ COMMITED
MS SQL Server 2012	READ COMMITED	SERIALIZABLE
Oracle 11g	READ COMMITED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITED	SERIALIZABLE
SAP HANA	READ COMMITED	SERIALIZABLE
ScaleDB 1.02	READ COMMITED	READ COMMITED
VoltDB	SERIALIZABLE	SERIALIZABLE