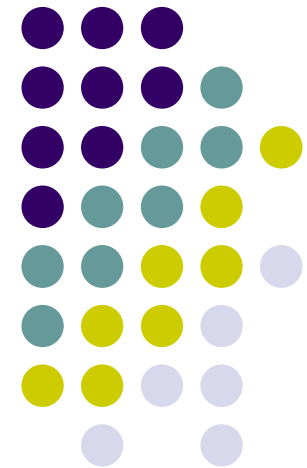
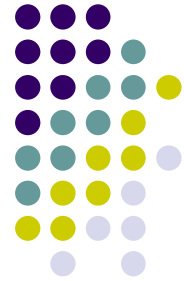


Transaction Management Overview

Book: Chapter 16 & 17



Transactions



- Transactions are „atomic“ (parts of) user programs (Example: bank transfer – reads/writes two tables)
- Concurrent execution of user programs is essential for good DBMS performance.
 - Disk operations and CPU operations can be performed concurrently.
 - Because disk accesses are frequent and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.

Issues to be discussed ...



- What is a valid Transaction? – ACID Properties
- Concurrency control:
 - **Interleaved** transactions and schedules
 - Problems (**anomalies**) with interleaved transactions
 - Systematic **detection of anomalies**
 - Avoidance of anomalies by means of **locking**
 - **Deadlocks**
- Log-files and checkpoints.

Transaction view of a DBMS



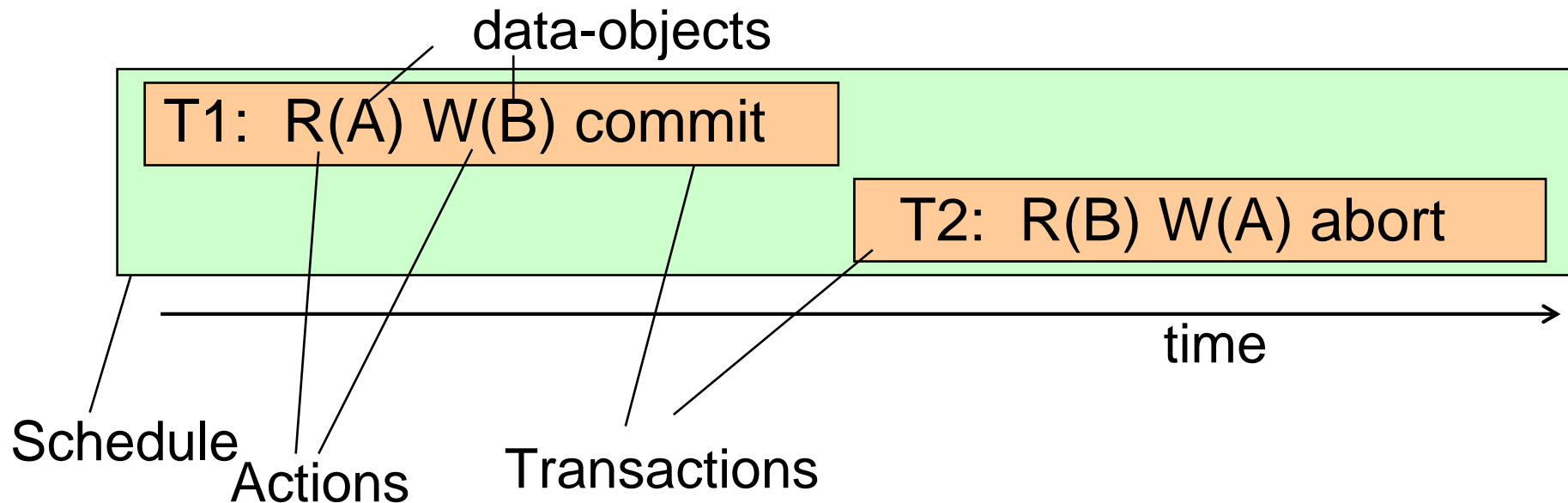
A user's program may carry out many operations on the data retrieved from the database, but:

*A DBMS is only concerned about what **data** is read/written from/to the database.*

Transactions, Actions, Schedules



- A transaction (*Xact*) is the DBMS's abstract view of a (part of a) user program: a sequence of reads and writes, that ends with commit or abort.



Atomicity of Transactions



- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic*. That is, a user can think of an Xact as always executing all its actions in one step, or not executing any actions at all.
 - DBMS *logs* **all** actions so that it can *undo* the actions of aborted transactions and *redo* the actions of committed transactions.



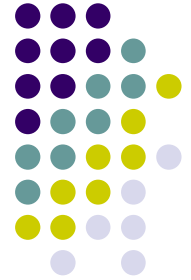
ACID Properties of Xact's

- Transactions should be **atomic**:
“Either all actions are carried out or none are.”
- Transactions should be **consistent**:
“Starting from a consistent DB state Xacts should result in a DB consistent state”
- Transactions should be **isolated or self-contained**:
“Users should understand the Xact. without considering other Xacts.”
- Transactions should be have a **durable** effect
“The effect of committed Xact's should persist, even after the system crashes.”

User's responsibility



- Each transaction must leave the database in a consistent state if the DBS is consistent when the transaction begins!
 - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
 - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

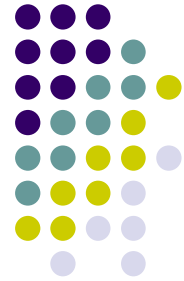


Transactions in SQL

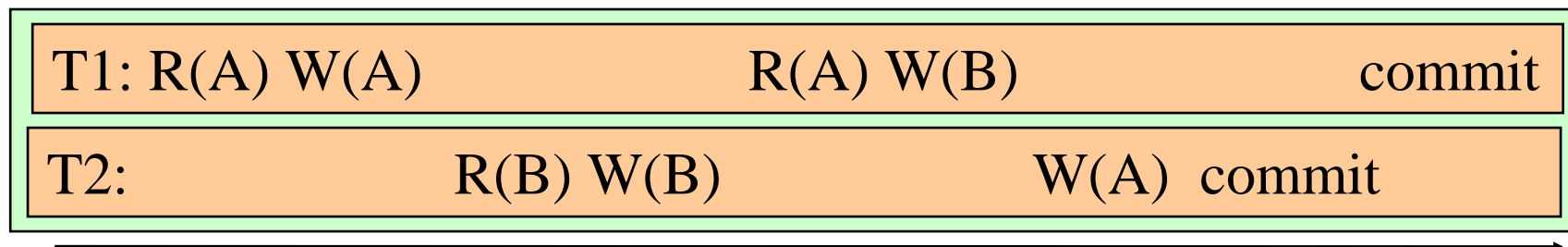
- COMMIT makes permanent any database changes you made during the current transaction. Until you commit your changes, other users cannot see them.
- ROLLBACK ends the current transaction and undoes any changes made since the transaction began.

```
insert into R values (1, 2); rollback;  
insert into R values (3, 4); commit;
```

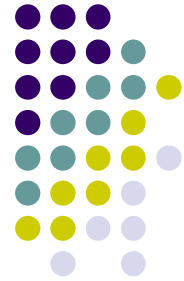
Concurrency in a DBMS



- Users submit transactions and can think of each transaction as executing by itself.
- Concurrency is achieved by the DBMS, which *interleaves* actions (reads/writes of DB objects) of various transactions.



Two examples of interleaved schedules

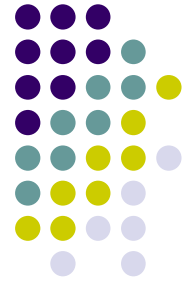


- Interleaved schedules can save time:

T1	R(A)	R(B)	process(A, B)	W(A)	commit
T2		R(B)	W(B)		commit

- But may cause inconsistencies (*anomalies*), as next examples show ...

Example: Banking



- Consider two transactions (*Xacts*):

T1: $A=A+100$, $B=B-100$ commit

T2: $A=1.06*A$, $B=1.06*B$ commit

Intuitively, the first transaction is transferring \$100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.

There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running **serially** in some order.

Example (Contd.)



- Consider a possible interleaving (schedule):

T1:	A=A+100,	B=B-100	cmt.
T2:	A=1.06*A,	B=1.06*B	cmt.

This is OK. But what about:

T1:	A=A+100,	B=B-100	cmt.
T2:	A=1.06*A, B=1.06*B		cmt.

Abstract view of a database:

T1:	R(A) W(A)	R(B) W(B)	cmt.
T2:	R(A) W(A) R(B) W(B)		cmt.

“Dirty Read” or Reading Uncommitted Data (WR Conflict)!



Example: Internet shop



T1: R(A) A=A-1 W(A) ... commit

T2: R(A) A=A-1 W(A) ... commit

1. A' may denote the number of items in a store, T1 and T2 remove one item each
2. Number of items should be reduced by two, after T1 and T2 committed



Example: Internet Shop



Serial execution of T1 and T2 is O.K.

T1: R(A) A=A-1 W(A) cmt.

T2: R(A) A=A-1 W(A) cmt.

But what about this schedule?

T1: R(A) A=A-1 W(A) cmt.

T2: R(A) A=A-1 W(A) cmt

Abstract view of the DBMS

T1: R(A) W(A) cmt.

T2: R(A) W(A) cmt.

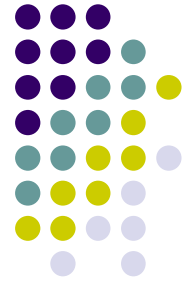
Unrepeatable Read (RW Conflict), or 'Lost update'

Comparing Schedules/ Serializability



- Serial schedule: Schedule that does not interleave the actions of different transactions.
- Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- Serializable schedule: A schedule that is equivalent to *some* serial execution of the committed transactions.
(**Note**: If each transaction preserves consistency, every serializable schedule preserves consistency.) **Why???**

Anomalies with Interleaved Execution



- Reading Uncommitted Data (WR Conflicts, “dirty read”):

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), Cmt.	

- Unrepeatable Reads (RW Conflicts):

T1:	R(A),	R(A), W(A), Cmt.
T2:	R(A), W(A), Cmt.	

- Overwriting uncommitted data (WW Conflicts):

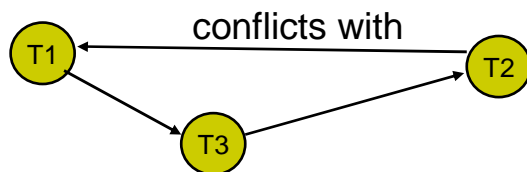
T1:	W(A),	W(B), Cmt.	
T2:	W(A), W(B),		Cmt.

Blind writes (write without reading)

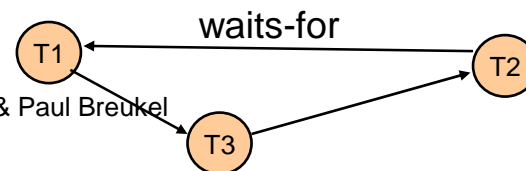
Detecting anomalies systematically – general idea



- Schedules can be checked automatically for **conflict serializability** by means of a **precedence graph**
- **Locking** protocols can guarantee conflict serializable schedules
- However, locking may introduce deadlocks, which can be detected with **waits-for graphs**



Databases: Michael Emmerich & Paul Breukel



Conflict Serializable Schedules



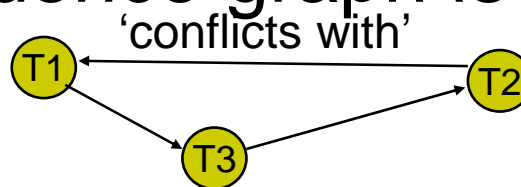
- Two actions a_1 and a_2 are said to be conflicting, iff
 - they belong to different transactions,
 - both approach the same object, e.g. A
 - and at least one of a_1 and a_2 is a write operation.
- Two schedules are **conflict equivalent** if:
 - They involve the same actions of the same transactions.
 - Every pair of conflicting actions is ordered the same way.
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule.
- A **conflict serializable** schedule is always **serializable**, but not vice versa. (see dia nr 22)

Precedence Graph



- Precedence graph: One node per Xact; edge from T_i to T_j if T_j reads/writes an object last read/written by T_i and at least one of the two operations is a write operation.

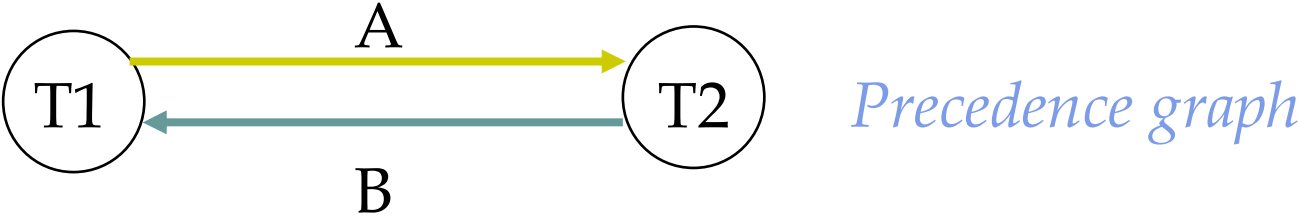
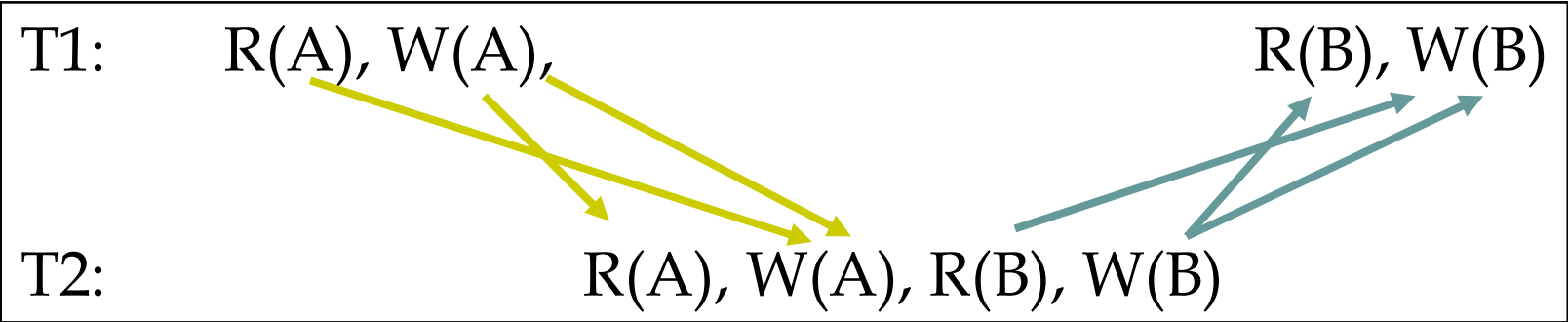
Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic.



Example



- A schedule that is not conflict serializable:



Serializable vs. Conflict serializable



T1:	W(A)	W(A) W(B), commit
T2:	W(A) R(A) W(B) commit	

T2;T1 would be the equivalent serial schedule for this schedule, which is not **conflict** serializable.

Review: Strict 2PL (Bernstein, Goodman, Hadzilacos)



- [Strict Two-phase Locking \(Strict 2PL\) Protocol:](#)
 1. Each Xact must obtain a *S (shared)* lock on object before reading, and an *X (exclusive)* lock on object before writing.
 2. All locks held by a transaction are released when the transaction completes (abort or commit).
 3. A Xact cannot lock an object with a shared lock if an exclusive lock on that object is held by another Xact.
 4. A Xact cannot lock an object with an exclusive lock if a (shared or exclusive) lock on that object is held by another Xact.
- Note: Read locks can be held by several transactions.
- If a write lock has been set, no read lock or write lock can be set and vice versa.
- Strict 2PL allows only schedules whose precedence graph is acyclic → it enforces conflict serializable schedules.



2 Examples: strict 2PL

Without locking

T1: R(A) R(B)	W(A) cmt.
T2: R(B) W(B) cmt.	

With strict 2PL

T1: S(A) R(A) S(B) R(B)	X(A) W(A) cmt.
T2: S(B) R(B)	X(B) W(B) cmt.

Without locking

T1: R(A) W(A)	R(B) W(B) cmt.
T2: R(A) W(A) R(B) W(B)	cmt.

With strict 2PL

T1: S(A) R(A) X(A) W(A) S(B) R(B) X(B) W(B) cmt.	
T2: S(A) R(A) X(A) W(A) ...	

Deadlocks



- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
 - Deadlock prevention
 - Deadlock detection

Without locking

T1: R(A) W(A)	R(B) W(B) cmt.
T2: R(B) W(B) R(A) W(A)	cmt.

With locking

T1: S(A) R(A) X(A) W(A)	waiting for release of B ...
T2: S(B) R(B) X(B) W(B)	waiting for release of A ...

Deadlock Detection



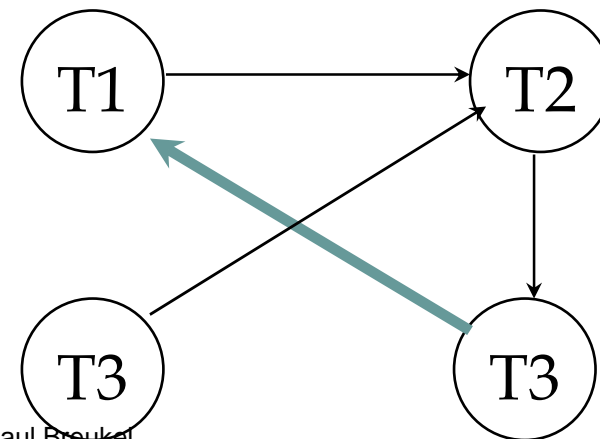
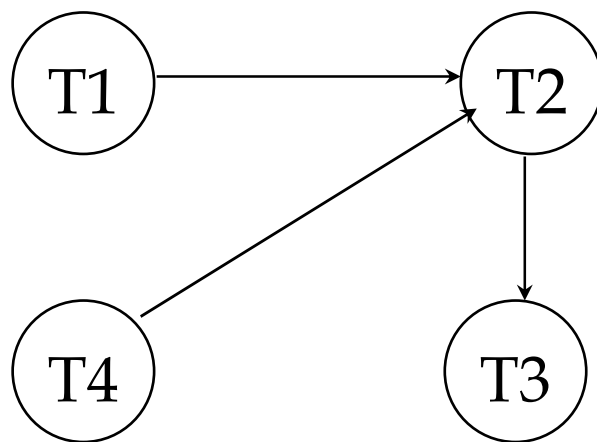
- Create a **waits-for graph**:
 - Nodes are transactions
 - There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Periodically check for cycles in the waits-for graph.

Deadlock Detection (Continued)



Example:

T1: S(A), R(A), S(B)
 T2: X(B), W(B) X(C)
 T3: S(C), R(C) X(A)
 T4: X(B)



Deadlock Prevention



- Assign priorities based on priorities. Assume T_i wants a lock that T_j holds. Two policies are possible:
 - Wait-Die: If T_i has higher priority, T_i waits for T_j ; otherwise T_i aborts.
 - Wound-wait: If T_i has higher priority, T_j aborts; otherwise T_i waits.
- If a transaction re-starts, make sure it has its original priority.
- Important: each transaction has a unique priority.

Tuning the schedulers using locking



- Release Locks, which are not necessary.
- Divide long transactions into many short transactions, if possible. (atomic!)
- Use special techniques for system backups and long reads. (Oracle!)
- Control granularity of locks (Record-, Page- or Table-level Locking).
- Schema updates only if systems activity is low (locked systems catalog can be a “bottleneck”)
- Reduce deadlock intervals for 2PL (check cyclic wait-graphs).

Other 2PL locking mechanisms

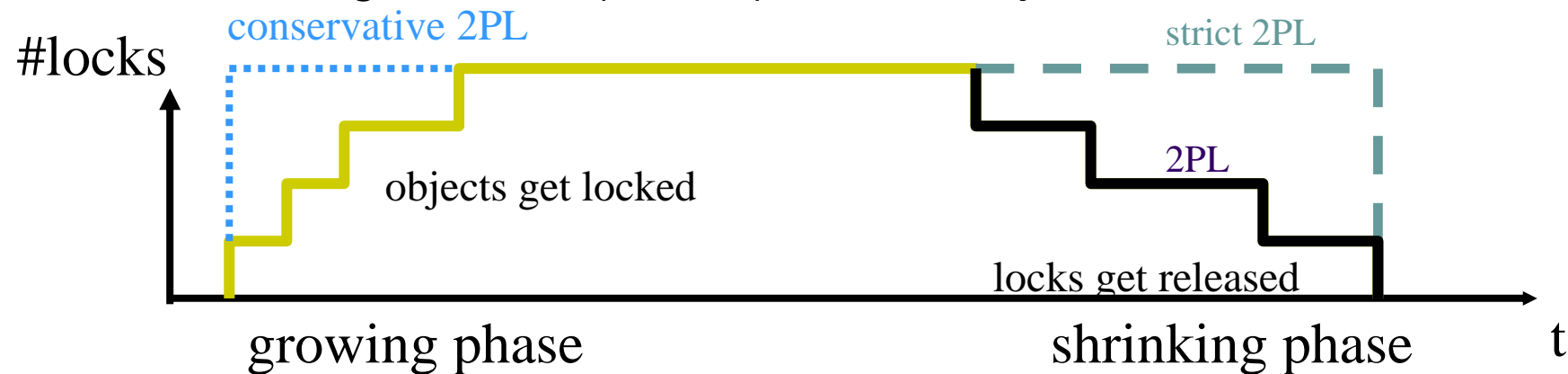


- Conservative 2PL
 - Like 2PL, but **all** locks have to be set already at the beginning of the transaction.
 - Deadlocks are avoided, but throughput might be decreased.
 - Dirty reads possible.
- Simple 2PL
 - Locks can be released before commit/abort (not a good idea).

(simple) Two-Phase Locking (2PL)



- Two-Phase Locking Protocol
 - Each Xact must obtain a *S* (*shared*) lock on object before reading, and an *X* (*exclusive*) lock on object before writing.
 - A transaction can not request additional locks once it releases any locks.
 - If an Xact holds an *X* lock on an object, no other Xact can get a lock (*S* or *X*) on that object.





Why strict 2Phase Locks?

- T1 locks A ($X(T1,A)$)
 - T1 releases A
 - Now, T2 locks A
 - Transaction T2 ends ($commit(T2)$)
 - T1 abort
-
- Not only T1 but also T2 has to be made undone (Cascading abort) to make it **recoverable**.
 - If T2 already committed! → Violation of **durability** principle (see ACID).

Aborting a Transaction



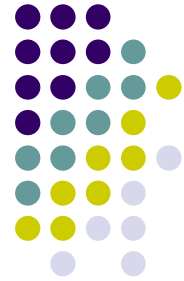
- If a transaction T_i is aborted, all its actions have to be undone. Not only that, if T_j reads an object last written by T_i , T_j must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time (strict 2PL).
 - If T_i writes an object, T_j can read this only after T_i commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

The Log



- The following actions are recorded in the log:
 - *Ti writes an object*: the old value and the new value.
 - Log record must go to disk before the changed page itself! (write ahead logging).
 - *Ti commits/aborts*: a log record indicating this action.
- Log is often *duplexed* and *archived* on stable storage.

Crash recovery

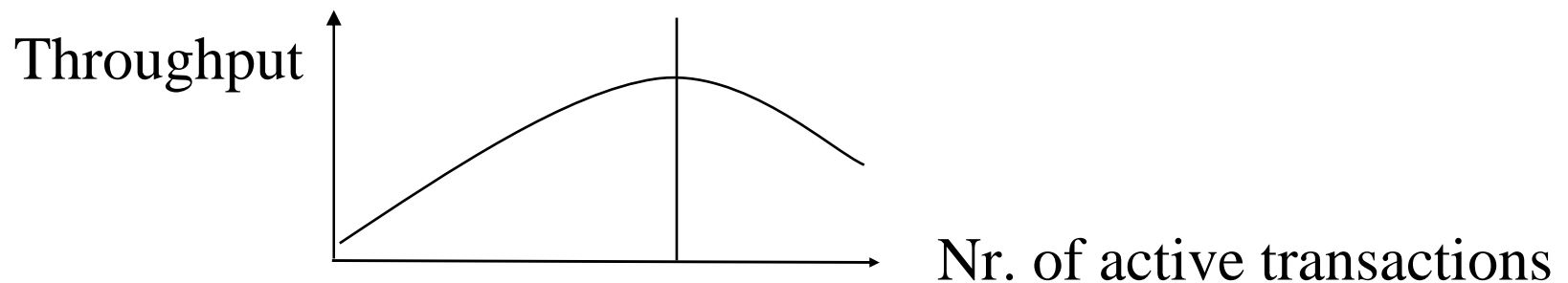


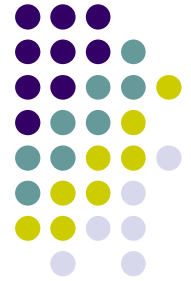
- In order to achieve recoverability, a write ahead log file (WAL) is used.
 - Every write is written to the WAL before it is committed.
 - After it has been committed another entry note is put to the log.
- Checkpoints store consistent database states at certain timestamps:
 - At each checkpoint all transactions have been committed (i.e. there are no active transactions)
 - Rollback algorithm has to trace back only to checkpoints (not further back), even if cascading appears (like in simple 2PL)



Further issues

- Granularity of locks, *locks and indexes*
- Crash recovery algorithms
- Performance of CC
- Lock thrashing





Alternatives to 2PL protocols

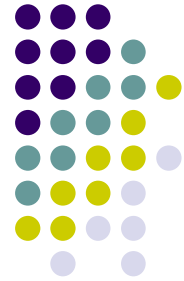
- Strict 2PL most frequently used protocol.
- Concurrency control with timestamps.
- Optimistic concurrency control
 - Proceed transaction in buffer
 - Before commit: validate if there were conflicts.
 - If so: abort.
 - Otherwise: commit.
 - More efficient, if conflicts are very unlikely.
 - No deadlocks!

Alternatives to 2PL protocols



- Multi Version Concurrency Control (MVCC):
 - Each write action results in a new version of the data object.
 - Old versions can concurrently be read by other transactions.
 - Concurrency control is much more difficult!
 - Oracle: long read transactions.

Summary



- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
 - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
 - *Consistent state*: Only the effects of committed Xacts seen.