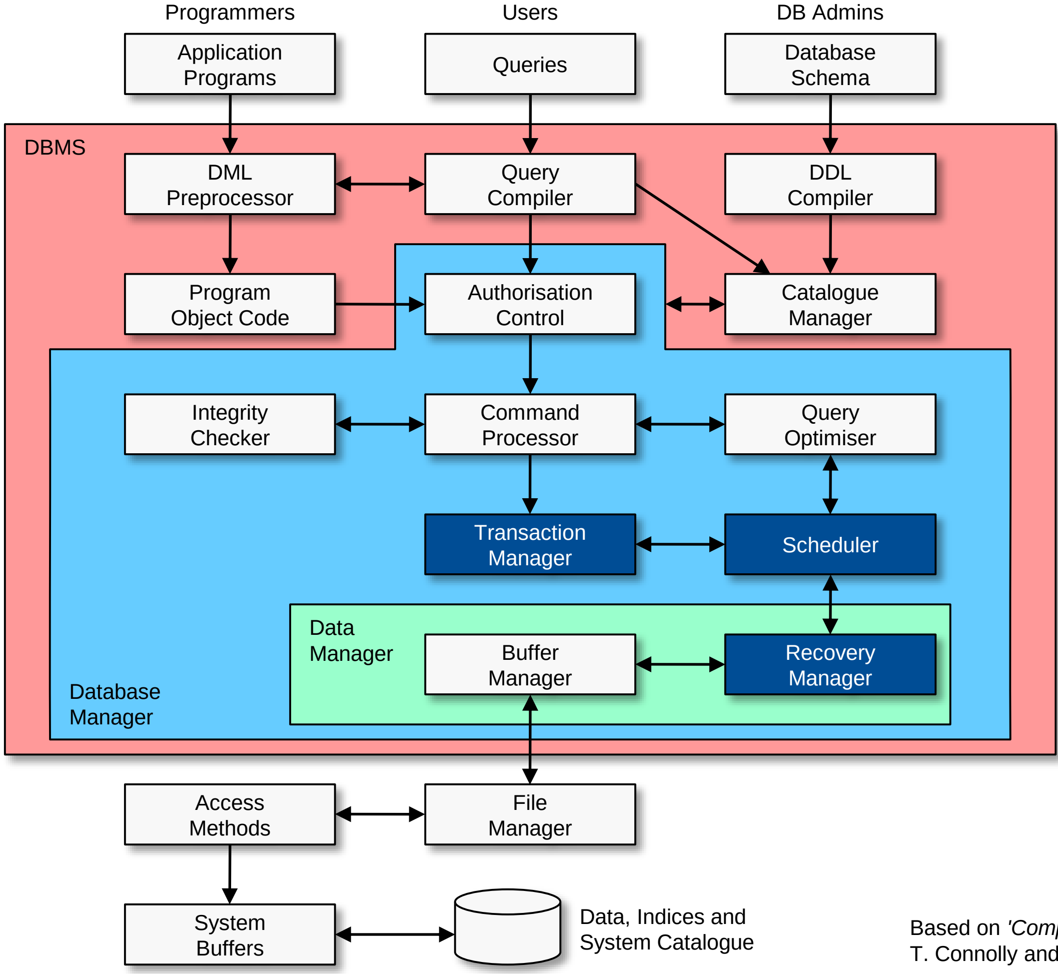
Introduction to Databases
Transaction Management

Prof. Beat Signer

Department of Computer Science
Vrije Universiteit Brussel

http://www.beatsigner.com

Context of Today's Lecture



Based on *'Components of a DBMS'*, Database Systems, T. Connolly and C. Begg, Addison-Wesley 2010

Transaction

Ariane Peeters wants to transfer 700 Euro from her bank account (314-229) to her landlord's account (889-752)

EXEC transfer(314-229, 889-752, 700)

```
CREATE PROCEDURE transfer(accountA CHAR(10), accountB CHAR(10), amount DECIMAL(12,2))
BEGIN
  DECLARE currentBalance DECIMAL(12,2);
  SELECT balance INTO currentBalance
  FROM account                                                R(account)
  WHERE account.accountNumber = accountA;

  IF (currentBalance > amount) THEN
    UPDATE account
    SET account.balance = balance – amount                    W(account)
    WHERE account.accountNumber = accountA;

    UPDATE account
    SET account.balance = account.balance + amount            W(account)
    WHERE account.accountNumber = accountB;
  ENDIF
END
```

Transaction ...

- A transaction is a *sequence of operations* that form a *single unit of work*
- A transaction is often initiated by an application program
  - begin a transaction
    - `START TRANSACTION`
  - end a transaction
    - `COMMIT` (if successful) or `ROLLBACK` (if errors)
- A transaction T_i *transforms one consistent* database state *into another consistent* database *state*
  - during the execution of T_i the DB *may be temporarily inconsistent*
- Either the whole transaction must succeed or the effect of all operations has to be undone (*rollback*)

Transaction ...

- There are two main transaction issues
  - *concurrent execution* of multiple transactions
  - *recovery* after hardware failures and system crashes
- In many SQL implementations, each SQL statement is a transaction on its own
  - this default behaviour can be disabled
  - SQL:1999 introduced `BEGIN ATOMIC ... END` blocks
  - see earlier SQL and Advanced SQL lectures for more details
- To preserve the integrity of data, the DBMS has to ensure that the so-called *ACID properties* are fulfilled for any transaction

ACID Properties

- *Atomicity*
  - either all operations of a transaction are reflected in the database or none of them (*all or nothing*)
- *Consistency*
  - if the database was is a consistent state before the transaction started, it will be in a *consistent state after* the *transaction* has been executed
- *Isolation*
  - if transactions are executed in parallel, the *effects of an ongoing transaction* must *not* be *visible* to other transactions
- *Durability*
  - after a transaction finished successfully, its *changes are persistent* and will not be lost (e.g. on system failure)

Money Transfer Example Revisited

- Transaction to transfer money from account A to B

```
1. start transaction
2. read(A)
3. A = A-700
4. write(A)
5. read(B)
6. B = B+700
7. write(B)
8. commit
```

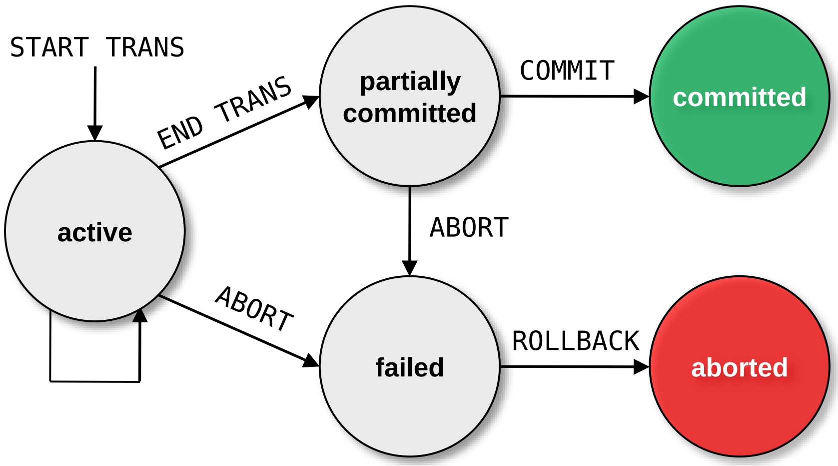
- Atomicity
  - if the transaction fails after step 4 but before step 8, the updates on A should not be reflected in the database (rollback)
- Consistency
  - the sum of A and B should not be changed by the transaction

Money Transfer Example Revisited ...

- Isolation
  - if another transaction is going to access the partially updated database between step 4 and 7, it will see an inconsistent database (with a sum of A and B which is less than it should be)
- Durability
  - once the money has been transferred from A to B (commit), the effect of the transaction must persist
    - the only way to "undo" a committed transaction is to execute a *compensating transaction*

Transaction States

- *Active*
  - initial state; transaction is in this state while executing
- *Partially committed*
  - after the last statement has been executed
- *Committed*
  - after successful completion
- *Failed*
  - after discovery that a normal execution is no longer possible
    - logical error (e.g. bad input), system error (e.g. deadlock) or system crash
- *Aborted*
  - after the rollback of a transaction



Transaction Management

- Transaction Manager
  - ensures that we proceed from one consistent state to another consistent state (*database consistency*)
  - ensures that transactions will not violate integrity constraints
- Scheduler
  - provides a specific strategy for the execution of transactions and the corresponding *concurrency control*
  - avoids or resolves conflicts during concurrent data access
- Recovery Manager
  - *restore the database* to the state it was in before a failure occurred (e.g. due to software bug or hardware problem) while executing one or multiple transactions

Scheduler

- *Serial execution* of transactions
  - each operation within a transaction can be executed atomically
  - any *serial execution* of a set of transactions $T_1,..., T_n$ by different users is regarded as a *correct result*
- *Parallel execution* of transactions
  - improves the throughput and resource utilisation as well as the average response time
  - too much parallelism can lead to wrong results
    - e.g. dirty reads, lost updates, phantoms, ...
  - the scheduler has to choose the appropriate *concurrency control scheme* to avoid problems during parallel execution

Schedule

- A *schedule S* specifies the *chronological order* in which the *operations* of concurrent transactions are executed
  - a schedule for the transaction $T_1,..., T_n$ must contain all operations of these transactions
  - the schedule must *preserve* the *order* of the operations *in* each *individual transaction*

Example Schedules

- Let transaction T_1 transfer 700 Euro from account A to B and T_2 transfer 10% of the balance from A to B
  - critical are the read (R) and write (W) operations
- Schedule 1

T_1	R(A)	A=A-700	W(A)	R(B)	B=B+700	W(B)							
T_2							R(A)	t=A*0.1	A=A-t	W(A)	R(B)	B=B+t	W(B)

  - serial schedule where T_1 is followed by T_2
- Schedule 2

T_1	R(A)	A=A-700	W(A)					R(B)	B=B+700	W(B)			
T_2				R(A)	t=A*0.1	A=A-t	W(A)				R(B)	B=B+t	W(B)

  - non-serial schedule but *equivalent* to Schedule 1

Example Schedules ...

- Schedule 3

T_1	R(A)	A=A-700						W(A)	R(B)	B=B+700	W(B)		
T_2			R(A)	t=A*0.1	A=A-t	W(A)	R(B)					B=B+t	W(B)

  - this schedule *does not preserve the sum* of A and B and therefore leads to problems

Conflict Serialisability

- Two operations of transactions T_i and T_j form a *conflict pair* if *at least one* of them is a *write operation*
- If a schedule S can be transformed into a schedule S' by a series of *swaps of non-conflicting operations*, then S and S' are *conflict equivalent*
  - a conflict pair forces an order on the transactions
- A schedule S is *conflict serialisable* if it is conflict equivalent to a serial schedule

	$\mathtt{R_i}(x)$	$\mathtt{W_i}(x)$
$\mathtt{R_j}(x)$	✓	✗
$\mathtt{W_j}(x)$	✗	✗

conflict pairs

Conflict Serialisability Example

- Transactions
  - T_1: $R_1(x)$ $W_1(x)$ $R_1(z)$
  - T_2: $R_2(y)$ $R_2(x)$ $W_2(y)$
  - T_3: $R_3(y)$ $R_3(z)$ $W_3(z)$
- Schedule S

$R_1(x)$	$W_1(x)$	$R_2(y)$	$R_2(x)$	$R_3(y)$	$R_3(z)$	$W_2(y)$	$W_3(z)$	$R_1(z)$

- Conflict pairs
  - $<W_1(x),R_2(x)>$, $<R_3(y),W_2(y)>$ and $<W_3(z),R_1(z)>$
- Conflict equivalent schedule

$R_3(y)$	$R_3(z)$	$W_3(z)$	$R_1(x)$	$W_1(x)$	$R_1(z)$	$R_2(y)$	$R_2(x)$	$W_2(y)$

Conflict Serialisability Example ...

- Transactions
  - T_1: $R_1(x)$ $W_1(x)$
  - T_2: $W_2(x)$
- Schedule S

$R_1(x)$ | $W_2(x)$ | $W_1(x)$

- Conflict pairs
  - $<R_1(x),W_2(x)>$ and $<W_2(x),W_1(x)>$
- This schedule is not conflict serialisable!

Testing for Serialisability

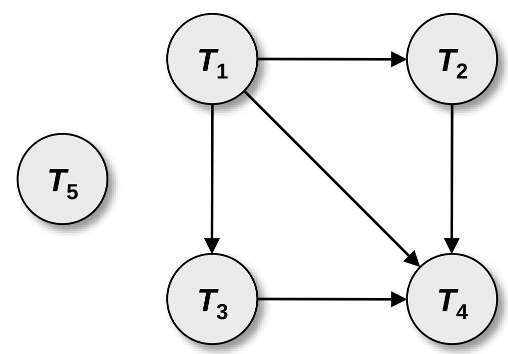
- Check if there is a serial schedule with the same ordering of the conflict pairs as the given schedule S
- Construct a *precedence graph* $G=(V,E)$ for S
  - the vertices V are represented by the transactions $T_1,\ldots, T_n$
  - there is an edge from T_i to T_j if there exists a conflict pair $<x,y>$ in S with $x \in T_i$ and $y \in T_j$ and x is preceding y
  - *a schedule S is serialisable if its precedence graph is acyclic*
- Algorithm to find and equivalent serial schedule S'
  - construct the precedence graph for the schedule S
  - perform a *topological sorting* of the graph
    - randonly choose a vertex with no incoming edges and remove the vertex and its outgoing edges from S (add its operations to S')
    - repeat the vertex removal until there are no more vertices or a cycle occurs

Conflict Serialisability Example

- Schedule S

$R_2(x)$	$R_1(y)$	$R_1(z)$	$R_5(v)$	$R_5(w)$	$R_2(y)$	$W_2(y)$	$W_3(z)$
$R_1(u)$	$R_4(y)$	$W_4(y)$	$R_4(z)$	$W_4(z)$	$R_1(u)$	$W_1(u)$	

- Conflict pairs
  - $<R_1(y),W_2(y)>$, $<R_1(y),W_4(y)>$, $<R_2(y),W_4(y)>$, $<W_2(y),R_4(y)>$, $<W_2(y),W_4(y)>$, $<R_1(z),W_3(z)>$, $<R_1(z),W_4(z)>$, $<W_3(z),R_4(z)>$, $<W_3(z),W_4(z)>$

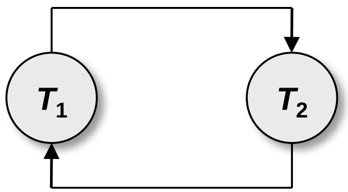


- Serialisable schedule $T_5 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$
  - note that there is more than one serialisable schedule for S

Conflict Serialisability Example ...

- Schedule S

$R_1(x)$	$W_2(x)$	$W_1(x)$



- Conflict pairs
  - $<R_1(x),W_2(x)>$ and $<W_2(x),W_1(x)>$
- There is *no serialisable schedule* for S since the precedence *graph has a cycle*

Concurrency Control

- Different *concurrency control schemes* can be used to ensure that the isolation property is ensured when multiple transactions are executed in parallel
- The concurrency control schemes for implementing serialisation in an online system include
  - lock-based protocols
  - validation-based protocols
    - timestamp ordering
    - optimistic concurreny control
  - graph-based protocols

Lock-based Protocols

- One way to ensure serialisability is to require that data items can only be accessed in a mutually exclusive manner
- The DBMS has to offer a mechanism to lock a specific data object x for a given transaction T_i and mode m
  - lock(T_i, x, m)
  - unlock(T_i, x)
- A transaction has to *request a lock in the appropriate mode* for a data object and can only proceed if the scheduler grants the lock

Lock-based Protocols ...

- At any given time, there can never be two trans-actions with incompatible locks on the same data object
  - the second transaction has to wait until the object in unlocked
  - DBMS puts the waiting transactions into specific queues
- Current DBMSs implement two types (modes) of locks
  - *exclusive-mode lock* (X)
    - read and write access to the data object
  - *shared-mode lock* (S)
    - read-only access
    - at any time several shared-mode locks can be held simultaneously

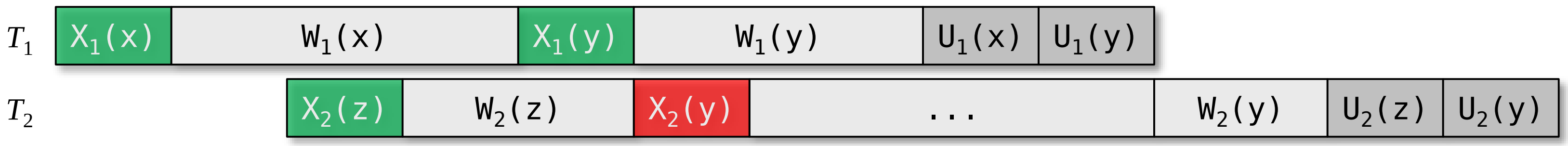
	S	X
S	✓	✗
X	✗	✗

lock compatibility matrix

  - lock is only granted if there is no other transaction that is already waiting for a lock on the same data object (to *prevent starvation*)

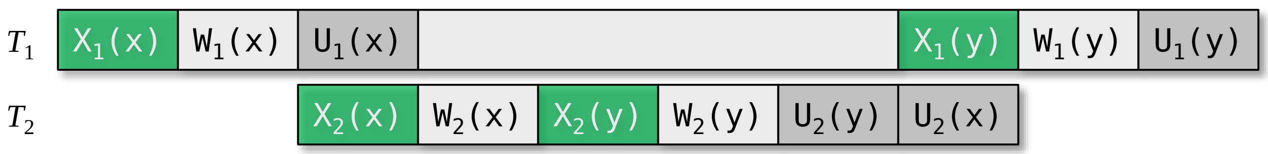
Locking Example

- Transactions
  - T_1: $W_1(x)$ $W_1(y)$
  - T_2: $W_2(z)$ $W_2(y)$
- Lock
  - exclusive lock $X_i(x)$ and unlock $U_i(x)$
- Schedule S

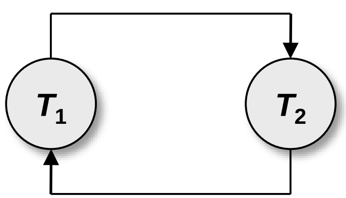


Locking

- Note that *locking* on its own *does not guarantee the serialisability* of a set of transactions $T_1,\ldots, T_n$
- Schedule S



- Precedence graph

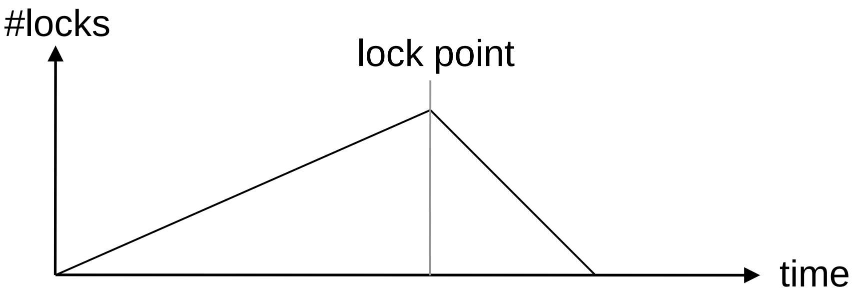


The graph has a cycle and therefore the schedule is not serialisable!

- We need more than just locking → *two-phase locking*

Two-Phase Locking (2PL)

- The two-phase locking protocol is based on two rules
- Rule 1
  - a data object has to be locked (exclusive or shared lock) before it can be accessed by a transaction (*growing phase*)
- Rule 2
  - as soon as a transaction unlocks its first data object, it cannot acquire any further locks (*shrinking phase*)

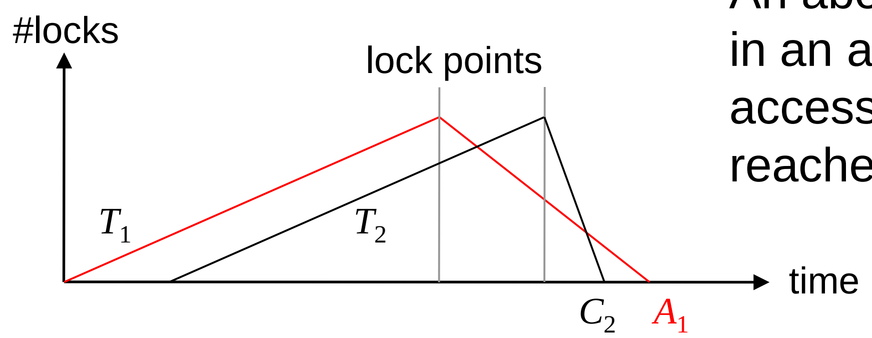


Two-Phase Locking (2PL)

- The 2PL protocol *guarantees serialisable schedules*
- Problems of the 2PL protocol
  - if we want to use the 2PL protocol, we have to know for each transaction T_i when no further locks will be necessary (*lock point*)
    - not very realistic to have this kind of a priori knowledge
  - the 2PL protocol is *not deadlock free*
  - potential problems in the case of an abort/rollback of an operation (*cascading rollbacks*)

2PL Cascading Rollback Problem

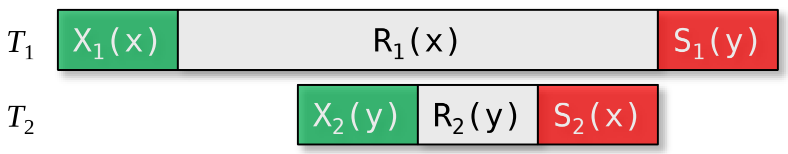
- The 2PL protocol is not suited to handle transactions that are aborted since *cascading rollbacks* may occur
- Cascading rollback
  - transaction T_i ends with an abort/rollback operation A
  - this might trigger a previously committed transaction T_j to be aborted too!
- Abort example



An abort A_1 of transaction T_1 can result in an abort of transaction T_2 since it got access to "shared" data objects after T_1 reached its lock point

2PL Deadlock Problem

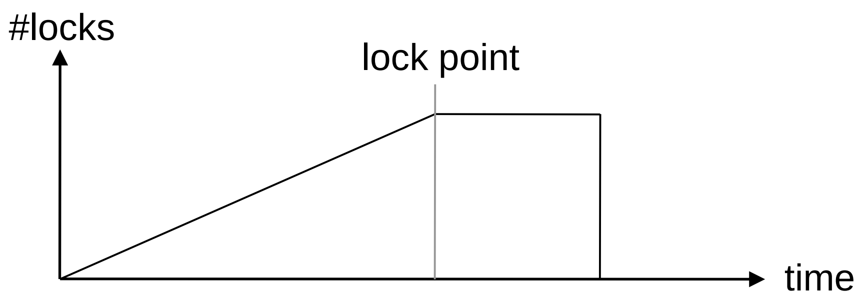
- Two transactions T_i and T_j might wait in a cycle for a lock held by the other transaction
- Example



- The scheduler has to periodically check whether such cycles (deadlocks) exist
  - use a directed *wait-for graph* to model the dependencies between transactions
  - if a deadlock occurs (cycle in the wait-for graph), the scheduler has to *reset* one of the participating *transactions*

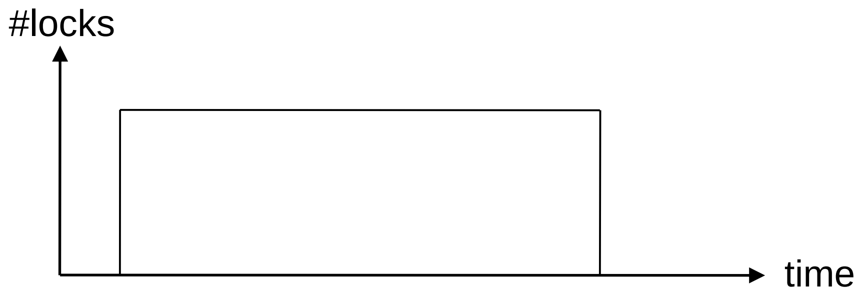
Strict Two-Phase Locking (S2PL)

- The strict two-phase locking protocol is based on two rules
- Rule 1
  - a data object has to be locked (exclusive or shared lock) before it can be accessed by a transaction (*growing phase*)
- Rule 2
  - a transaction keeps all locks until the end of the transaction and releases them all at once (*commit/abort phase*)



Strict Two-Phase Locking (S2PL)

- The S2PL protocol guarantees serialisable schedules
  - S2PL *avoids cascading aborts* but it is still *not deadlock free*
- S2PL is implemented in every major database system
- S2PL can be implemented in a deadlock-free manner (*deadlock prevention*)
  - transaction has to aquire all necessary locks (*preclaiming of locks*) before the first operation is executed
  - *reduces potential concurrency* for long transactions



Locking Granularity

- Locking protocols such as 2PL or S2PL can be applied at various granularity levels
  - pages/blocks
    - commonly used
  - relations
    - very coarse and restrictive
  - tuples
- There is a trade-off between concurrency and overhead
- Small granularity
  - higher level of potential concurrency but larger locking overhead
- Large granularity
  - less potential for concurrency but smaller locking overhead

Validation-based Protocols

- Rather than to prevent conflicts from the beginning, it is sufficient to detect them and resolve them
  - abort transaction in the case that a conflict is detected
  - works efficiently if the probability for conflicts is very low
- Example scheduling techniques
  - timestamp ordering
  - optimistic concurrency control

Timestamp Ordering

- Each transaction T_i gets a unique timestamp $TS(T_i)$ assigned
  - e.g. based on system clock or a logical counter
- Timestamp ordering rule
  - if operation $o_{i,m}(x)$ is in conflict with operation $o_{j,n}(x)$ and $o_{i,m}(x)$ is part of transaction T_i whereas $o_{j,n}(x)$ is part of T_j, then they have to be ordered $o_{i,m}(x) < o_{j,n}(x)$ if $TS(T_i) < TS(T_j)$
  - the *timestamp order defines* the *serialisation order*
- For each access of a data object the scheduler has to check whether a later transaction (larger timestamp) has already accessed the object
  - `W-TS(x)`: largest timestamp of transaction writing *x*
  - `R-TS(x)`: largest timestamp of transaction reading *x*

Timestamp Ordering Scheduler

- T_i wants to read object x
  - $\mathrm{TS}(T_i) < \mathrm{W\text{-}TS}(x)$
    - T_i wants to read old data $\rightarrow$ *reset T_i*
  - $\mathrm{TS}(T_i) \geq \mathrm{W\text{-}TS}(x)$
    - *permit read operation* and update $\mathrm{R\text{-}TS}(x)$
- T_i wants to write object x
  - $\mathrm{TS}(T_i) < \mathrm{R\text{-}TS}(x)$
    - there is a newer transaction that already read $x \rightarrow$ *reset T_i*
  - $\mathrm{TS}(T_i) < \mathrm{W\text{-}TS}(x)$
    - T_i wants to write an obsolete value $\rightarrow$ *reset T_i*
  - $\mathrm{TS}(T_i) \geq \mathrm{R\text{-}TS}(x)$ and $\mathrm{TS}(T_i) \geq \mathrm{W\text{-}TS}(x)$
    - *permit write operation* and update $\mathrm{W\text{-}TS}(x)$

Optimistic Concurrency Control (OCC)

- Assumes that there will not be many conflicts
- Transactions are executed with the explicit risk of abortion
  - in *snapshot isolation* each transaction has a private workspace
- Three phases of a transaction T_i
  - *reading and execution phase*
    - T_i reads objects of the database (read set of T_i) and writes private versions (write set of T_i)
  - *validation phase*
    - before writing the private versions to the database a conflict analysis is performed
  - *writing phase*
    - if the validation was positive the private versions are written to the disk

Recovery

- The *recovery manager* has to ensure that the *atomicity* and *durability* properties are preserved in the case of a system failure
- Different types of system failures
  - *transaction failure*
    - *logical error*
      - internal transaction problems (e.g. bad data input or data not found)
    - *system error*
      - system in an undesirable state (e.g. deadlock)
  - *system crash*
    - software bug or hardware malfunction not affecting the non-volatile storage
  - *disk failure*
    - content loss on non-volatile storage (e.g. data transfer error or head crash)

Log-based Recovery

- After a system failure we should be able to return to a state where ongoing transactions have either been sucessfully completed or had no effect on the data at all
- To support *undo* and *redo operations*, we must write logging information to a stable storage before modifying the database
  - atomicity
    - undo based on logging information
  - persistency
    - redo based on logging information
  - redo and undo operations must be *idempotent*
    - executing them multiple times has the same effect as executing them once

Log-based Recovery ...

- A *log file* consists of a *sequence of log records* which can be of the following types
  - *start* of transaction T_i
    - <T_i start>
  - transaction T_i *updates* the value V_1 of data item X_j to value V_2
    - <T_i, X_j, V_1, V_2>
  - *commit* of transaction T_i
    - < T_i commit>
  - *abort* of transaction T_i
    - < T_i abort>
- We can either perform an *immediate* or a *deferred database modification*

Immediate Modification Technique

- Allows *uncommited* database *modifications* while the transaction is still in its active state
- Before a transaction T_i starts, we write the corresponding *start record* to the log
- Each write operation is preceded by the corresponding *update record* in the log
- When the transaction T_i partially commits, we write the corresponding commit record to the log
- To support concurrent transactions we further have to ensure that there are no conflicting update operations (e.g. by using S2PL)

Logging Example

Transaction T_1

```
read(A)
A = A-700
write(A)
read(B)
B = B+700
write(B)
```

Transaction T_2

```
read(C)
C = C-500
write(C)
```

- The two transactions might result in the following log file (annotated with the database state)

Log file

```
&lt;T1 start&gt;
&lt;T1, A, 2000, 1300&gt;
&lt;T2 start&gt;
&lt;T2, C, 1300, 800&gt;
&lt;T2 commit&gt;
&lt;T1, B, 1800, 2500&gt;
&lt;T1 commit&gt;
```

Database state

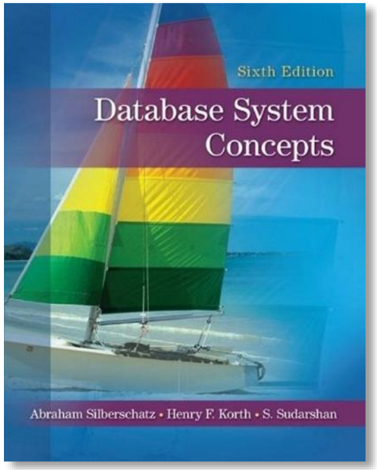
```


A = 1300

C = 800

B = 2500
```

Restart Recovery

- After a crash the system scans the log file and constructs two lists of transactions
  - *redo-list*
    - contains each transaction T_i for which a $<T_i \text{ commit}>$ log record exists
  - *undo-list*
    - contains each transaction T_i for which no $<T_i \text{ commit}>$ log record exists
- The system then performs the following two steps
  (1) scan the log file *in reverse order* and for each log record of a transaction T_i in the undo-list perform an undo operation
  (2) scan the log file *in forward order* and for each log record of a transaction T_i in the redo-list perform a redo operation
- To avoid a scan of the entire log file, special *checkpoint* operations can be performed periodically

Homework

- Study the following chapters of the *Database System Concepts* book
  - chapter 14
    - sections 14.1-14.10
    - Transactions
  - chapter 15
    - sections 15.1-15.11
    - Concurrency Control
  - chapter 16
    - sections 16.1-16.10
    - Recovery System



Exercise 10

- Query Processing and Query Optimisation

References

- A. Silberschatz, H. Korth and S. Sudarshan, *Database System Concepts* (Sixth Edition), McGraw-Hill, 2010

Next Lecture

NoSQL Databases