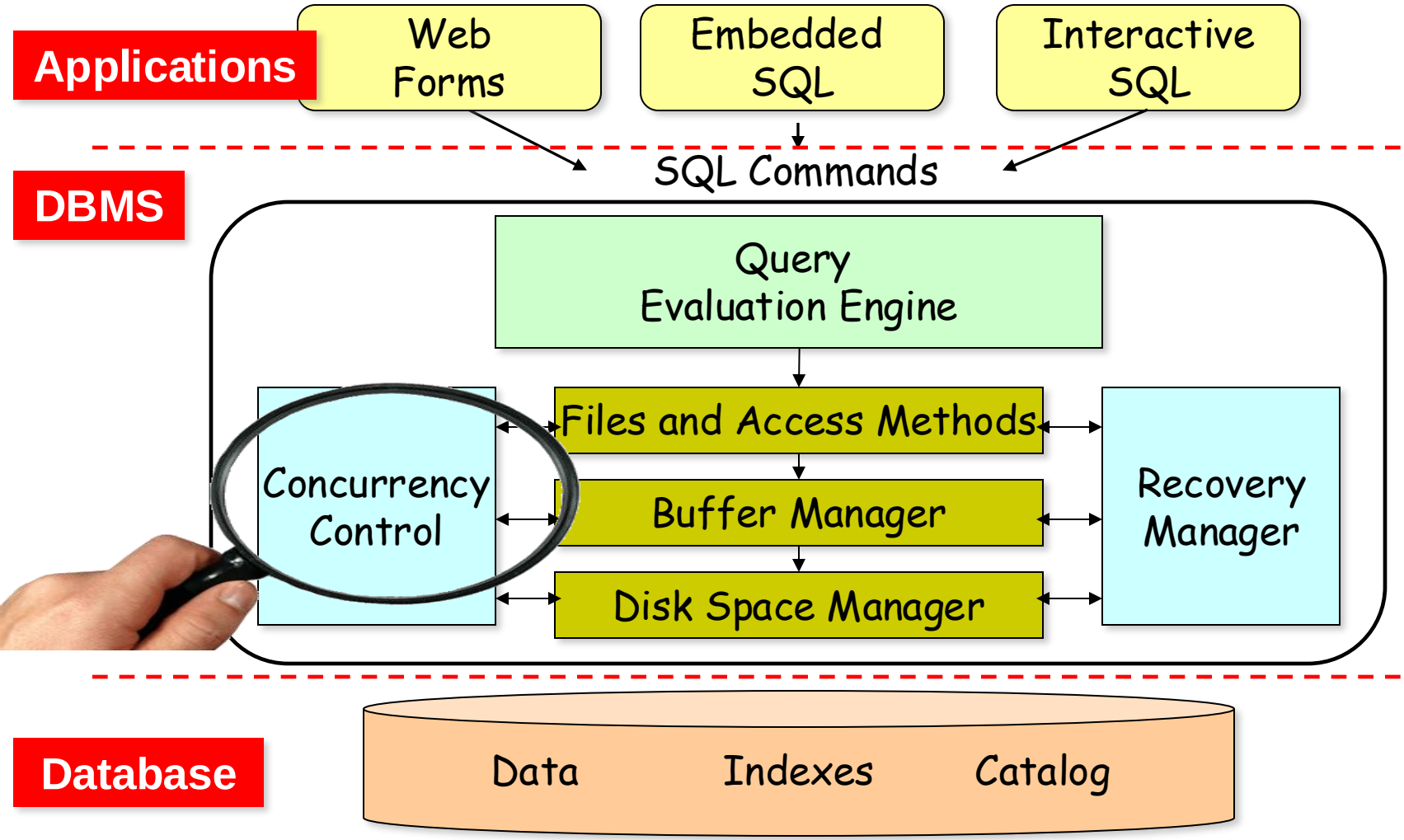
Database Management System (DBMS)



Queries and Transactions

- Queries: requests to the DBMS to retrieve data from the database
- Updates: requests to the DMBS to insert, delete or modify existing data
- Transactions: logical grouping of query and update requests to perform a task
  - Logical unit of work (like a function/subroutine)

Two Views of Transactions

```
set transaction read write
insert into Enrolment values ('S1234', 'CS545');
select * from Enrolment; ☺
commit
```

1. Application Programmer's View

   Start

   Sequence of **SQL statements**

   Commit or Rollback

2. System Developer's View

   Start

   Sequence of **Reads and Writes**

   Commit or Abort

Transactions

T_1: `UPDATE` Accounts `SET` balance = balance + 100 `WHERE` client=7

T_2: `UPDATE` Accounts `SET` balance = balance + 500 `WHERE` client=7

- Assume that initially balance = $1000
- What is the balance after executing T_1 & T_2?
  - Should be $1600

However things might go wrong!!

Transactions

T_1: `UPDATE` Accounts `SET` balance = balance + 100 `WHERE` client=7

T_2: `UPDATE` Accounts `SET` balance = balance + 500 `WHERE` client=7

- Update (balance) =

  Read (balance); Modify (balance); Write (balance)

- Again, assume that initially balance = $1000
- What happens if the two transactions are executed **concurrently** and they both issue Read (balance) at the same time?
  - If T_1 finishes last; balance = $1100
  - If T_2 finishes last, balance = $1500
  - And both values are incorrect!

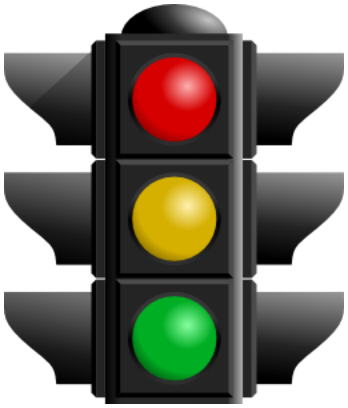
ACID Properties

Atomicity

Consistency

I solation

Durability



ACID Properties

- **Atomicity**

  Either all the operations associated with a transaction happen or none of them happens

- **Consistency**

  It satisfies the integrity constraints on the database at the transaction's boundaries

- **Isolation**

  The result of the execution of concurrent transactions is the same as if transactions were executed serially

- **Durability**

  The effects of completed transactions become permanent surviving any subsequent failures

Consistency

T_1: `INSERT INTO Enrolment VALUES ('S1234', 'CS545');`

T_2: `UPDATE Accounts SET balance = balance - 100 WHERE client=7`

- **Consistency:** It satisfies the integrity constraints on the database at the transaction's boundaries
  - E.g., 'S1234' or 'CS545' must exist
  - E.g., balance is not allowed to be negative
- Mechanism: **Integrity Constraints (ICs)**
  - *Foreign key constraint …*
  - *Checks, Assertions, Triggers …*

Atomicity

$T_{transfer}$

```
UPDATE Accounts SET balance = balance – 400 WHERE client=7
```


```
UPDATE Accounts SET balance = balance + 400 WHERE client=9
```

- **Atomicity:** Either **all** the operations associated with a transaction happen **or none** of them happens
- Each transaction is **"all-or-nothing"**; never left half-done
- Mechanism: **Logging**
  - *Recovery…*

Durability

T_1: `UPDATE` Accounts `SET` balance = balance + 100 `WHERE` client=7

- **Durability:** The effects of completed transactions become permanent surviving any subsequent failures
- If system crashes after transaction T_1 commits, all effects of T_1 remain in database
- Challenge: DBMS moves data between memory and disk all the time and a crash can happen while data is still in memory
- Mechanism: **Logging**
  - *Recovery…*

Isolation

T_1: `UPDATE` Accounts `SET` balance = balance + 100 `WHERE` client=7

T_2: `UPDATE` Accounts `SET` balance = balance + 500 `WHERE` client=7

- **Isolation:** The result of the execution of concurrent transactions is the same as if transactions were executed **serially**
- **Serializability:** Operations may be interleaved, but execution must be equivalent to some sequential (**serial**) order of transactions
  - E.g., T_1 followed by T_2, or T_2 followed by T_1
- Mechanism: **Concurrency Control**
  - *This lecture…*

ACID Properties

Property	Dealt with by
A, D	Recovery Techniques
I	**Concurrency Control Techniques**
C	Integrity Constraints

Two Views of Transactions

1. Application Programmer's View

```
Start
   Sequence of SQL statements
Commit or Rollback
```

2. System Developer's View

```
Start
   Sequence of Reads and Writes
Commit or Abort
```

Transaction Primitives

START (alias BEGIN, SET, DECLARE)

- Informs the system that a new transaction starts to execute.
- The transaction becomes *active*.

COMMIT

- Informs the system that a transaction has successfully completed its task.
- The system makes the effects of the transaction **durable**.
- After this the transaction is *committed*.

ROLLBACK (alias Abort)

- Informs the system that a transaction has terminated abnormally.
- The system must **undo** all the effects of the transaction.
- The transaction is *aborted*.

Two Views of transactions

1. Application Programmer's View

```
Start
   Sequence of SQL statements
Commit or Rollback
```

2. System Developer's View

```
Start
   Sequence of Reads and Writes
Commit or Abort
```

Database Model

- **A database -** collection of named data items
- Basic operations are **read** and **write**
  - **read_item(X)**: Reads the value of a database item named X into a program variable X.
  - **write_item(X)**: Writes the value of program variable X into the database item named X.
- **Granularity of data:**
  - A field, a record, a table, etc

Transaction: an Example in SQL

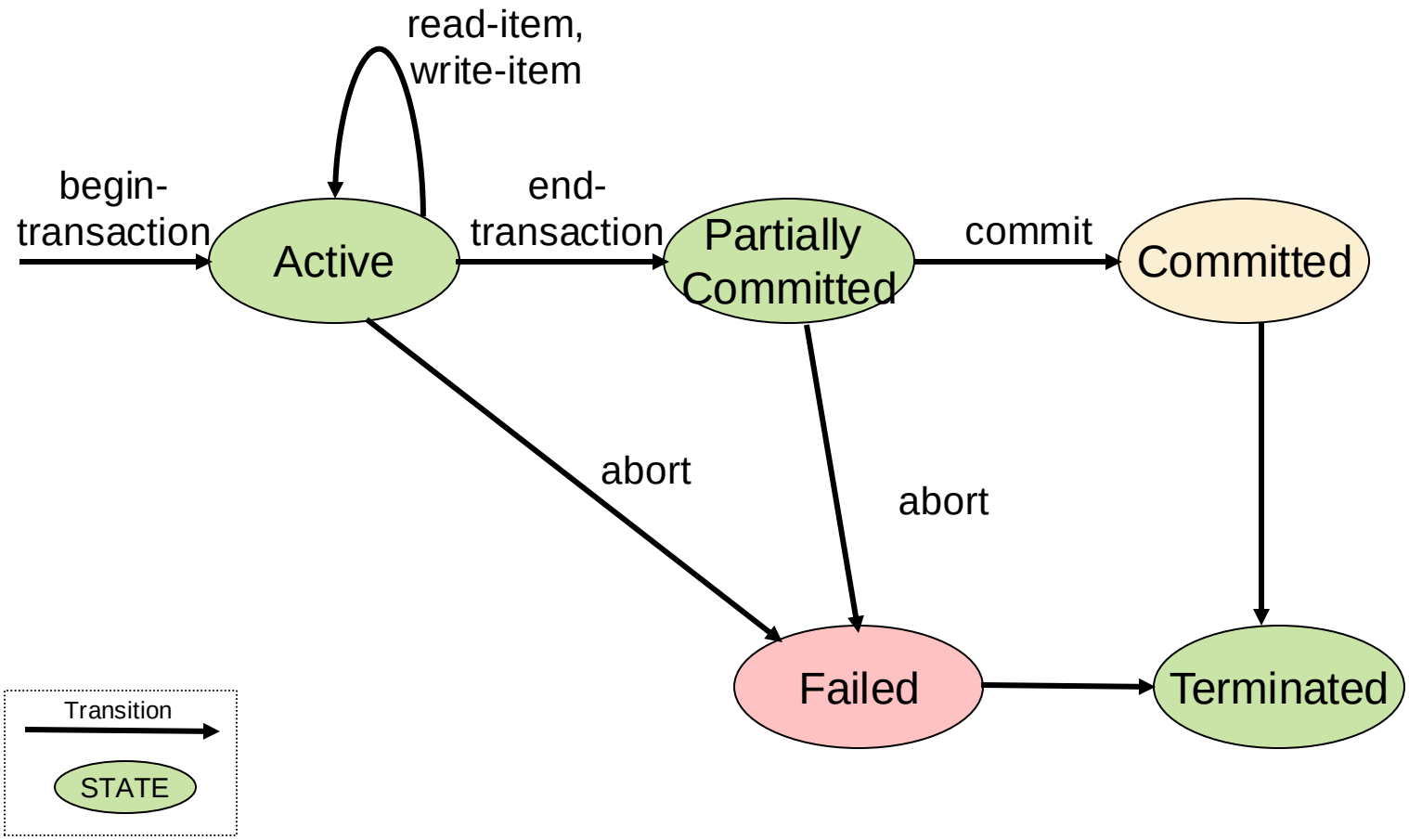
Transaction BUDGET_UPDATE

begin

```
UPDATE    PROJ
SET       BUDGET = BUDGET*1.1
WHERE     PNAME = “CAD/CAM”
```

end.

Transaction Lifecycle



...where things may go wrong?

When & Why Concurrency Control?

- Multi-User System:
  - **Many** users (transactions) access the system **concurrently**.
- Simple Solution
  - Execute them **in isolation**!
- But want to <u>enable</u> **concurrency** whenever it is safe:
  - **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing** in multiple CPUs.

When & Why Concurrency Control?

Transaction processing systems are large databases with multiple users executing database transactions

Multiple users are concurrently executing transactions

Transactions can have several data access operations, some of which could be accessing the same data item

Through multi-programming operating systems, the processes of users are interleaved

Concurrency Goal: Isolation

- As if each transaction is executed by **itself** in the system
- The effect of executing two **concurrent** transactions must be equivalent to these two transactions running **serially** in some order
- Example: If two transactions T_1 and T_2; Possible serial executions:
  - T_1 followed by T_2, or
  - T_2 followed by T_1
  - Both serial executions are correct

Serial Executions

Item X=100

T_1	T_2
read_item(X) X=X-10 write_item(X) commit	 read_item(X) X=X+5 write_item(X) commit

Item X=95

T_1	T_2
    read_item(X) X=X-10 write_item(X) commit	read_item(X) X=X+5 write_item(X) commit

Item X=95

Serial Executions

Item X=100

T_1	T_2
read_item(X) X=X+10 write_item(X) commit	read_item(X) X=X*2 write_item(X) commit

Item X=220

T_1	T_2
read_item(X) X=X+10 write_item(X) commit	read_item(X) X=X*2 write_item(X) commit

Item X=210

Concurrency vs. Anomalies

T_1	T_2
read_item(X)	
X=X-N	
	read_item(X)
	X=X+M
write_item(X)	
read_item(Y)	
	write_item(X)
Y=Y+N	
write_item(Y)	

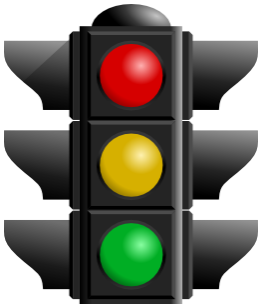
When transactions are executing concurrently, the execution order of operations form a **schedule**

Concurrent (i.e., interleaved) execution of transactions might cause **anomalies**!

Anomalies

- Question: why **concurrency control** is needed?
- Answer: to avoid the following anomalies:

1. The **lost update** problem
2. The **dirty read** problem
3. The **unrepeatable read** problem



Anomalies: Lost Update Problem

T_1	T_2
read_item(X)	
X=X-10	
	read_item(X)
	X=X+20
write_item(X)	
read_item(Y)	
	write_item(X)
Y=Y+10	
write_item(Y)	

Item X has an incorrect value because its update by T_1 is lost

Anomalies: Dirty Read Problem

T_1	T_2
read_item(X)	
X=X-10	
write_item(X)	
	read_item(X)
	X=X+20
	write_item(X)
	commit
read_item(Y)	
Y=Y+10	
write_item(Y)	
abort	

Transaction T_1 fails and must change the value of X back to its old value

Meanwhile T_2 has read the temporary incorrect value of X

Anomalies: Unrepeatable Read Problem

T_1	T_2
read_item(X)	
	read_item(X)
	X=X-1
	write_item(X)
read_item(X)	
X=X-1	
write_item(X)	

T_1 reads X twice, but X is changed between the reads

Jim Gray - The Godfather of Transactions!

https://www.youtube.com/watch?v=8gHxKQrxV8o

Schedules

T_1	T_2
read_item(X)	
X=X-N	
	read_item(X)
	X=X+M
write_item(X)	
read_item(Y)	
	write_item(X)
Y=Y+N	
write_item(Y)	

When transactions are executing concurrently, the execution order of operations form a **schedule**

Concurrent (i.e., interleaved) execution of transactions might cause **anomalies**!

Schedules

- When transactions are executing concurrently, the order of execution of operations from all transactions is known as a **schedule** (or **history**)
- A Schedule **S** of **n** transactions T_1, T_2, …, T_n is an ordering of the operations of the transactions
- For the purpose of concurrency control, we are mainly interested in the read (r) and write (w) operations, as well as commit (c) and abort (a) operations
  - Examples: *next 3 slides..*

Schedule: Example 1

T_1	T_2
read_item(X)	
X=X-N	
write_item(X)	
read_item(Y)	
Y=Y+N	
write_item(Y)	
	read_item(X)
	X=X+M
	write_item(X)

Why unacceptable in practice?

S: $r_1(x), w_1(x), r_1(y), w_1(y), c_1 , r_2(x), w_2(x), c_2$

Schedule: Example 2

T_1	T_2
read_item(X)	
X=X-N	
	read_item(X)
	X=X+M
write_item(X)	
read_item(Y)	
	write_item(X)
Y=Y+N	
write_item(Y)	

S: $r_1(x)$, $r_2(x)$, $w_1(x)$, $r_1(y)$, $w_2(x)$, c_2, $w_1(y)$, c_1

Schedule: Example 3

T_1	T_2
read_item(X)	
X=X-N	
write_item(X)	
	read_item(X)
	X=X+M
	write_item(X)
	commit
read_item(Y)	
Y=Y+N	
write_item(Y)	
abort	

S: $r_1(x), w_1(x), r_2(x), w_2(x), c_2, r_1(y), w_1(y), a_1$

Scheduling Transactions

- **Serial schedule:** Schedule that does not interleave the actions of different transactions
- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions
- **Result Equivalent schedules:** For any database state, two schedules S_1 and S_2 are equivalent if:
  - The state produced by executing S_1 is identical to the state produced by executing S_2

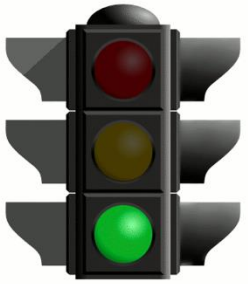
How to achieve serializability? Concurrency Control

Lock Based Concurrency Control

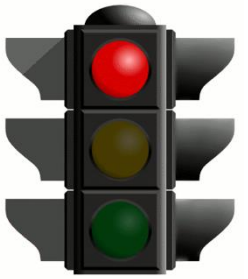
- Locking is the most common synchronization mechanism
- A **lock** is associated with each data item in the database
- A lock on item **“x”** indicates that a transaction is performing an operation (read or write) on **“x”**.

Lock Based Concurrency Control

- Transaction T_i can issue the following operations on item x:
  - **read_lock (x)**
    - x is read-locked by T_i
    - **shared** lock: other transactions are allowed to read x



  - **write_lock (x)**
    - x is write-locked by T_i
    - **exclusive** lock: single transaction holds the lock on x



  - **unlock (x)**

Lock Table

- How does a DBMS manage locks?
  - **Lock table:** Each entry in the lock table keeps information about a locked data item

Item Requested	Locks Granted	Queued Requests
X	<T1, read_lock>, <T2, read_lock>	<T3, write_lock>, <T5, write_lock>
Y	<T3, write_lock>	<T4, read_lock>, <T6, write_lock>

- What is the appropriate **granularity** of a data item?

Granularity of Locks

- Locking granularity is the **size** of the data item being locked. Examples:
  - Block
  - Table
  - Tuple (record)
  - Field in a tuple
  - A particular field of all tuples (column)
- The granularity of locks is irrelevant w.r.t. correctness, but it is important w.r.t. performance

Granularity of Locks

Locking Granularity

	Fine	Coarse
Lock conflict probability	?	?
Concurrency	?	?
Overhead: lock table size	?	?

Low or High?

Granularity of Locks

Locking Granularity

	Fine	Coarse
Lock conflict probability	Low	High
Concurrency	High	Low
Overhead: lock table size	High	Low

Sample Transactions

Item X=20 & Y=30

T_1	T_2
read_item(Y) read_item(X) X=X+Y write_item(X)	read_item(X) read_item(Y) Y=X+Y write_item(Y)

T_1 then T_2
X=50, Y=80

T_2 then T_1
X=70, Y=50

Sample Transactions

Item X=20 & Y=30

T_1	T_2
read_lock(Y)	
read_item(Y)	
unlock(Y)	
	read_lock(X)
	read_item(X)
	unlock(X)
	write_lock(Y)
	read_item(Y)
	Y=X+Y
	write_item(Y)
	unlock(Y)
write_lock(X)	
read_item(X)	
X=X+Y	
write_item(X)	
unlock(X)	

T_1	T_2
read_lock(Y)	
Y=30	
unlock(Y)	
	read_lock(X)
	X=20
	unlock(X)
	write_lock(Y)
	Y=30
	Y=50 (in T_2)
	Y=50 (on disk)
	unlock(Y)
write_lock(X)	
X=20	
X=50 (in T_1)	
X=50 (on disk)	
unlock(X)	

Sample Transactions

T_1	T_2
read_lock(Y)	
read_item(Y)	
unlock(Y)	
	read_lock(X)
	read_item(X)
	unlock(X)
	write_lock(Y)
	read_item(Y)
	Y=X+Y
	write_item(Y)
	unlock(Y)
write_lock(X)	
read_item(X)	
X=X+Y	
write_item(X)	
unlock(X)	

Item X=20 & Y=30

Result of Interleaved Schedule:
X=50, Y=50
(nonseiralizable!)

Item Y in T_1 was unlocked too early!

Locking in its own does not ensure serializability!

Need for protocol!

Basic Two Phase Locking (2PL)

- A scheduler following the 2PL protocol has two phases:

1. **A Growing phase**

   - Whenever the scheduler receives an operation on any item, it must acquire a lock on that item before executing the operation.
   - No locks can be released in this phase

2. **A Shrinking phase**

   - Once a scheduler has released a lock for a transaction, it cannot request any additional locks on any data item for this transaction.

Basic Two Phase Locking (2PL)

- Example:
  - **Transaction T:** a = r(x); w(y, a);

S_1:read_lock(x); a=r(x); write_lock(y); w(y, a); unlock(x); unlock(y);

S_2:read_lock(x); a=r(x); unlock(x); write_lock(y); w(y, a); unlock(y);

S_3:read_lock(x); a=r(x); write_lock(y); unlock(x); w(y, a); unlock(y);

- **Theorem**: Every 2PL schedule S is serializable.

Basic 2PL Example 1

T_1	T_2
read_item(X)	
X=X-10	
	read_item(X)
	X=X+20
write_item(X)	
read_item(Y)	
	write_item(X)
Y=Y+10	
write_item(Y)	

T_1	T_2
write_lock(X)	write_lock(X)
read_item(X)	read_item(X)
X=X-10	X=X+20
write_item(X)	write_item(X)
write_lock(Y)	unlock(X)
unlock(X)	
read_item(Y)	
Y=Y+10	
write_item(Y)	
unlock(Y)	

Follows the 2PL

No Lost Update problem!

Basic 2PL Example 2

- How about Unrepeatable Read Problem?

T_1	T_2
read_item(X)	
	read_item(X)
	X=X-1
	write_item(X)
read_item(X)	
X=X-1	
write_item(X)	

Basic 2PL Example 3

- How about Dirty Read Problem?

T_1	T_2
read_item(X)	
X=X-10	
write_item(X)	
	read_item(X)
	X=X+20
	write_item(X)
	commit
read_item(Y)	
Y=Y+10	
write_item(Y)	
abort	

Resolved by Strict 2PL

Basic 2PL Example 4

T_1	T_2
read_item(X)	
X=X-10	
	read_item(X)
	X=X+20
write_item(X)	
read_item(Y)	
	write_item(X)
Y=Y+10	
write_item(Y)	

T_1	T_2
read_lock(X)	read_lock(X)
read_item(X)	read_item(X)
X=X-10	X=X+20
write_lock(X)	write_lock(X)
write_item(X)	write_item(X)
read_lock(Y)	unlock(X)
read_item(Y)	
Y=Y+10	
write_lock(Y)	
unlock(X)	
write_item(Y)	
unlock(Y)	

Follows the 2PL, but can produce a deadlock!

Basic 2PL Example 5

T_1	T_2
read_item(Y) read_item(X) X=X+Y write_item(X)	read_item(X) read_item(Y) Y=X+Y write_item(Y)

T_1	T_2
read_lock(Y) read_item(Y) write_lock(X) unlock(Y) read_item(X) X=X+Y write_item(X) unlock(X)	read_lock(X) read_item(X) write_lock(Y) unlock(X) read_item(Y) Y=X+Y write_item(Y) unlock(Y)

Follows the two-phase locking protocol?

Basic 2PL Example 5

T_1	T_2
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
(waits for X)	write_lock(Y)
	(waits for Y)

But can produce a deadlock!

Basic 2PL Example 6

T_1	T_2
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
read_lock(X)	
X=X+Y	read_lock(Y)
	Y=X+Y

Does this schedule produce a deadlock?

No! at this point
A read-lock is shared; other transactions are allowed to read the data item

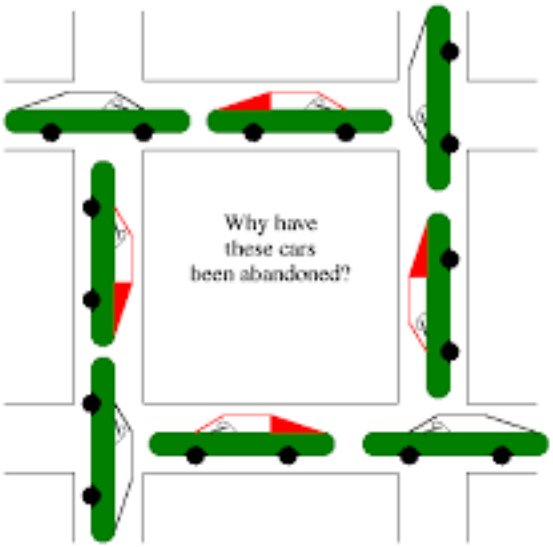
Two Phase Locking (2PL): Variants

- **Basic 2PL:**
  - T_x locks data items incrementally (growing phase)
  - This may cause deadlock: deal with it!
- **Conservative 2PL:**
  - Locks all desired data items before T_x begins execution
  - If any item cannot be locked, T_x does not lock any item
    - Waits until all items are available for locking
  - Deadlock-free protocol
  - Difficult to use in practice: hard to predeclare items!

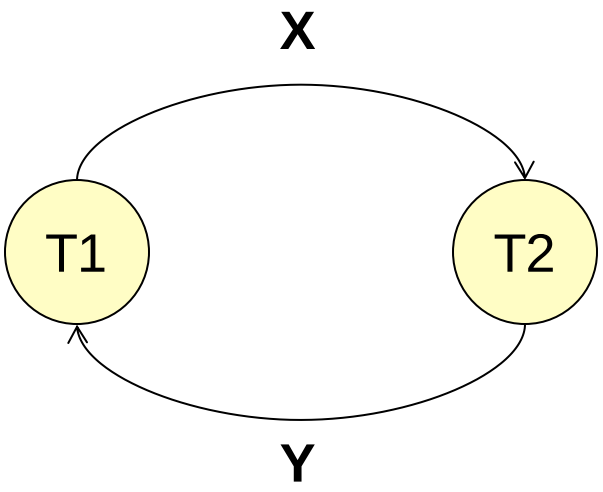
Dealing with Deadlocks

1. Deadlock Prevention
2. Deadlock Detection
3. Timeout

Next few slides…



Deadlock Prevention

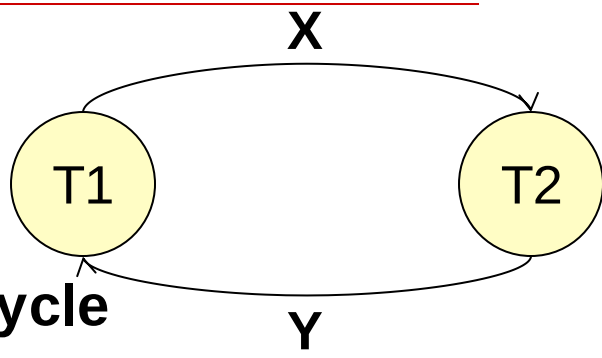


T_1	T_2
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
(waits for X)	write_lock(Y)
	(waits for Y)

- **Deadlock Prevention:**
  - T_x locks all its data items before it begins execution
  - Prevents deadlock since T_x never waits for a data item
  - The conservative two-phase locking uses this approach

Deadlock Detection

- **Deadlock Detection:**
  - Deadlocks are **allowed** to happen!
  - A wait-for-graph is used for detecting **cycle**



- **Wait-for-graph (WFG)**
  - Created using the **lock table**
  - As soon as a transaction is blocked, it is added to the graph
  - **Detect cycles:** T_i waits for T_j and T_j waits for T_i, then this creates a cycle!
  - One of the transactions in a cycle is **rolled back (aborted)**
    - **Which one?** *Next slide…*

Deadlock Detection: Victim Selection

- Factors to consider:
  - The **cost** of aborting a transaction
    - All updates must be **undone** (recall the “A” in ACID)
  - How long a transaction has been running?
  - How long it will take a transaction to finish?
  - How many **deadlocks** will be resolved if a particular transaction is aborted
    - Is the transaction in more than one cycle?
  - How many **times** this transaction was already aborted due to deadlocks?

Timeout

- **Guess** a deadlock rather than detecting it!
- The scheduler checks **periodically** if a transaction has been blocked for too long
  - In such a case, the scheduler assumes that the transaction is deadlocked and it is aborted
  - This method may incorrectly diagnose a situation to be a deadlock
    - The scheduler may make a mistake and abort a transaction that is waiting for another long transaction

Timeout

- **Advantage:** very simple algorithm

- **Fine tuning** of the timeout period:

  1. **Long timeout:**
     - Fewer mistakes by the scheduler, but
     - A deadlock may exist unnoticed for long periods
  2. **Short timeout:**
     - Quick deadlock detection, but
     - More mistakes are possible (aborting transactions not involved in a deadlock)

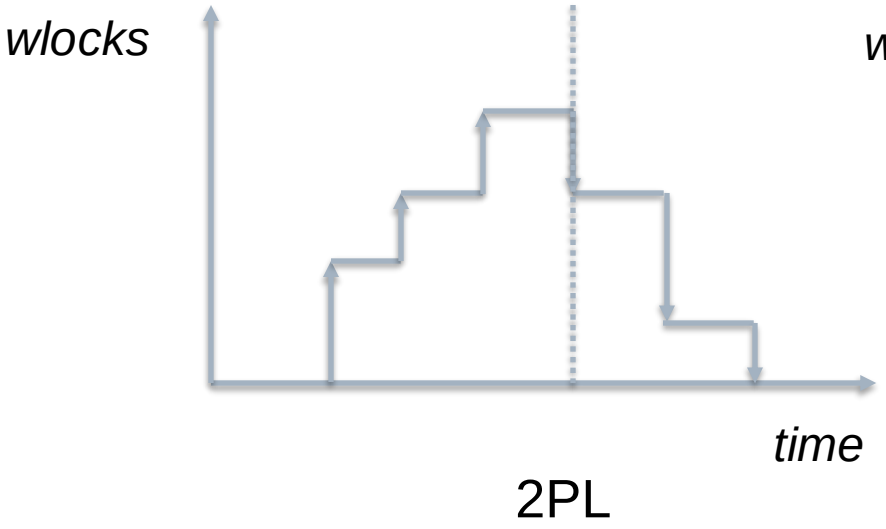
Need for Another 2PL Variant

- Transactions are aborted for different reasons:
  - Deadlocks, errors, etc.
  - All updates must be **undone** (recall the "A" in ACID)
  - What about other transactions that have already read those updates? (recall "**dirty**" read problem)

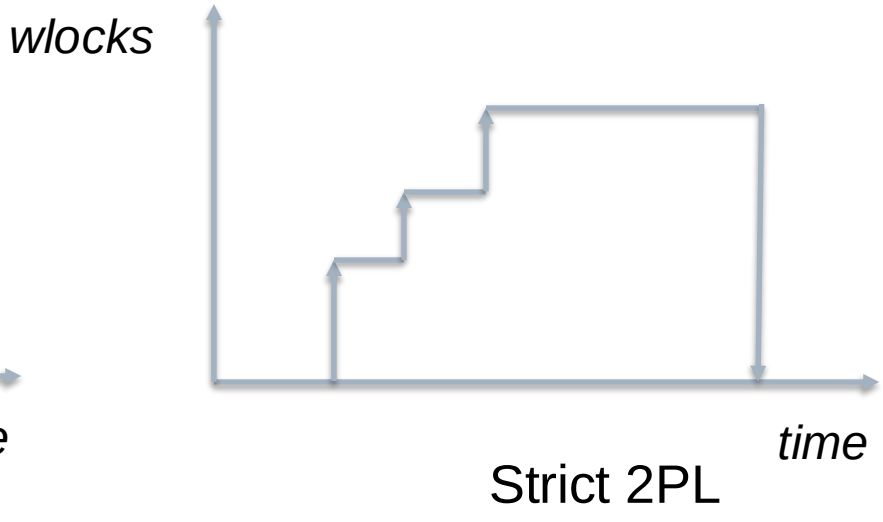
- **Strict 2PL:**
  - A stricter version of the Basic 2PL algorithm
  - Does not release exclusive (write) locks until the transaction terminates (commits or aborts)

Basic 2PL vs. Strict 2PL

- **Basic 2PL:**
  - Lock growing and shrinking
- **Strict 2PL:**
  - Does not release exclusive (write) locks until the transaction terminates (commits or aborts)



2PL



Strict 2PL

Advantages of the Strict 2PL Variant

- Is it deadlock free?
  - No!

- What is the advantage?
  - Strict 2PL simplifies the recovery process
    - Only rollback the aborted transaction
    - Other transactions are not affected since they couldn't read any updates made by the aborted transaction

Strict 2PL Example 1

T_1	T_2
read_item(X)	
X=X-10	
write_item(X)	
	read_item(X)
	X=X+20
	write_item(X)
	commit
read_item(Y)	
Y=Y+10	
write_item(Y)	
abort	

T_1	T_2
write_lock(X)	write_lock(X)
read_item(X)	read_item(X)
X=X-10	X=X+20
write_item(X)	write_item(X)
write_lock(Y)	unlock(X)
read_item(Y)	commit
Y=Y+10	
write_item(Y)	
unlock(X)	
unlock(Y)	
abort	

Follows the Strict 2PL

No Dirty Read problem!

Strict 2PL Example 2

- Example:
  - Transactions **T_1 and T_2**

$S = T_1:W(X), T_2:R(Y), T_1:R(Y), T_2:R(X), T_1:Commit, T_2:Commit$

- Is this schedule allowed under **Strict 2PL**?
  - **No:** T_2 would not be able to issue $T_2:R(x)$ before T_1 commits
- Is this schedule allowed under **Basic 2PL**?
  - **Yes:** T_2 can issue $T_2:R(x)$ if T_1 entered the shrinking phase and released the lock on X before T_2 reads it
- Question: What about **Conservative 2PL**?

Result Equivalent vs Conflict Equivalent

T_1	T_2
read_item(X)	
X=X*2	
	read_item(X)
	X=X*10
write_item(X)	
read_item(Y)	
	write_item(X)
Y=Y+N	
write_item(Y)	

Initially: X=0

Lost Update Problem!

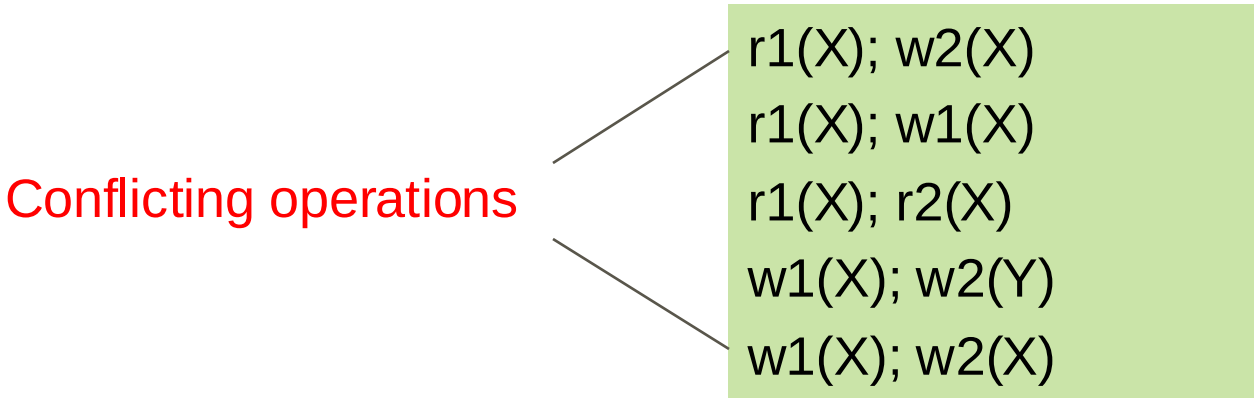
Yet schedule is result equivalent!

Scheduling Transactions

- **Result Equivalent** schedules**:** For **any** database state, two schedules S_1 and S_2 are equivalent if:
  - The state produced by executing S_1 is identical to the state produced by executing S_2
- **Conflict equivalent:** Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- **Conflict serializable (CSR):** A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

Conflicting Operations

- A conflict happens if we have two operations such that:
  1. They belong to two different transactions, and
  2. They both operate on the same data item,
  3. At least one of them is a write



Testing CSR

- Test for CSR by analyzing the precedence graph **SG(S)**, called a **serialization graph**, derived from schedule S.

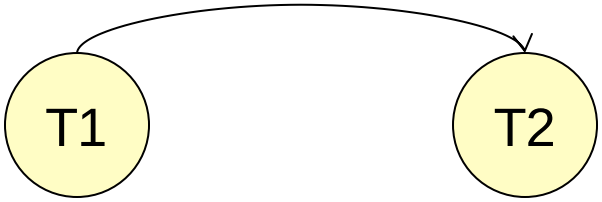
- An SG(S) is a directed graph in which:
  1. **Nodes** represent transactions in S;
  2. An **edge** $T_i \rightarrow T_j$, $i \neq j$, means that one of T_i's operations precedes and conflicts with one of T_j's operations in S.

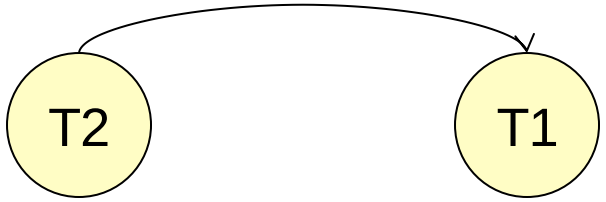
- Serializability Theorem:
  A schedule S is serializable iff SG(S) is **acyclic**.

Testing CSR: Example

T_1	T_2
read_item(X)	
X=X-N	
write_item(X)	
read_item(Y)	
Y=Y+N	
write_item(Y)	
	read_item(X)
	X=X+M
	write_item(X)

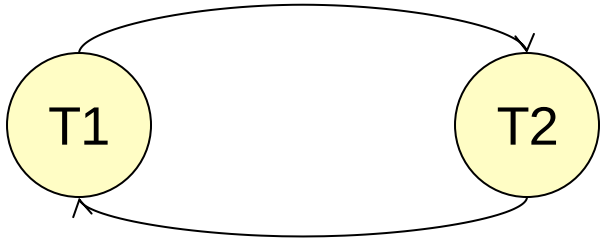
T_1	T_2
	read_item(X)
	X=X+M
	write_item(X)
read_item(X)	
X=X-N	
write_item(X)	
read_item(Y)	
Y=Y+N	
write_item(Y)	





Testing CSR: Example

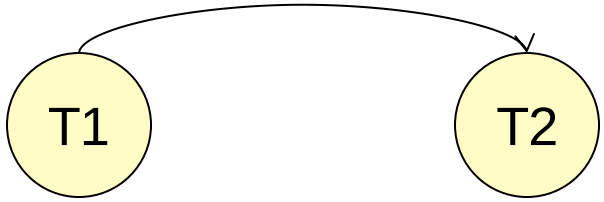
T_1	T_2
read_item(X)	
X=X-N	
	read_item(X)
	X=X+M
write_item(X)	
read_item(Y)	
	write_item(X)
Y=Y+N	
write_item(Y)	



S: $r_1(x)$, $r_2(x)$, $w_1(x)$, $r_1(y)$, $w_2(x)$, c_2, $w_1(y)$, c_1

Testing CSR: Example

T_1	T_2
read_item(X)	
X=X-N	
write_item(X)	
	read_item(X)
	X=X+M
	write_item(X)
read_item(Y)	
Y=Y+N	
write_item(Y)	



S: $r_1(x), w_1(x), r_2(x), w_2(x), c_2, r_1(y), w_1(y), c_1$

Testing CSR: Example

- Example:

  $S = r_1(x)\ r_2(x)\ w_1(x)\ c_1\ w_2(x)\ c_2$

- SG(S):

  $T_1 \rightarrow T_2$ because $w_1(x) <_S w_2(x)$ and $r_1(x) <_S w_2(x)$

  $T_2 \rightarrow T_1$ because $r_2(x) <_S w_1(x)$

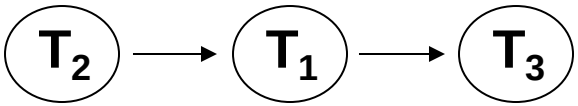
- Therefore, S is not serializable.

Finding The Equivalent Serial Schedule

- Make a topological sorting of the serialization graph.

$S = r_1(x)\ w_1(x)\ r_2(y)\ w_2(y)\ c_2\ r_1(y)\ w_1(y)\ c_1\ r_3(x)\ w_3(x)\ c_3$

SG(S)



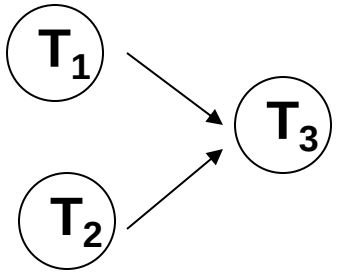
$S' = \underbrace{r_2(y)\ w_2(y)\ c_2}_{T_2}\ \underbrace{r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)\ c_1}_{T_1}\ \underbrace{r_3(x)\ w_3(x)\ c_3}_{T_3}$

Finding The Equivalent Serial Schedule

- Several serial orders can be obtained by topological sorting:

$S = r_1(x)\ w_1(x)\ r_3(x)\ w_2(y)\ c_1\ r_3(y)\ c_2\ w_3(y)\ c_3$

SG(S)



S': $T_1\ T_2\ T_3$ or $T_2\ T_1\ T_3$

Discussion

- Check if a schedule is serializable
  - Serialization graph
- Guarantee a schedule is serializable
  - Two-phase locking
- What happens in a real database system?