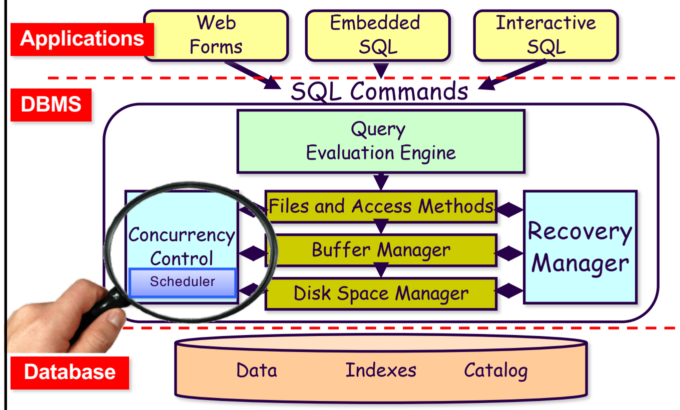
Transaction Processing:
Concurrency control

Database Management System (DBMS)



ACID Properties

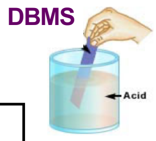
A tomicity

C onsistency

I solation

D urability

ACID Properties



Property	Dealt with by
A, D	Recovery Techniques
I	Concurrency Control Techniques
C	Checks, Assertions,Triggers Applications Programmers

Two Views of the System

```
set transaction read write
select * from Students
insert into Students values (777, ‘Jane’, ‘CS’)
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
```

Interleaved 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 T_1 and T_2 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***!

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** (one after the other)
- **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**

Concurrency Goal

- **Concurrency Goal:** Execute a sequence of SQL statements so they “appear” to be running in **isolation**
- Simple Solution
  - **Execute** them in isolation!

- But want to enable concurrency whenever it is **safe**:
  - High performance DBMS
  - Benefit from modern architectures

    (e.g., multicore processors, etc.)

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
    - The **phantom read**

Three Bad Dependencies

- ❑ ***Lost Update***: Read$_i$(X) Write$_j$(X) Write$_i$(X) sequence
    - Write-Write interaction (W-W)
- ❑ ***Dirty Data***: Write$_i$(X) Read$_j$ (X) sequence
    - Write-Read interaction (W-R)
- ❑ ***Unrepeatable Read***: Read$_i$(X) Write$_j$(X) Read$_i$(X) sequence
    - Read-Write interaction (R-W)
- ❑ These forms of inconsistency are the whole story.

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. and one of them is a write

Conflicting Operations

- ❑ Two operations ***conflict*** if it matters in which order they are performed
    - The order affects the results;
    - The order affects the state of the database.
- ❑ Non conflicting operations are called ***compatible***.
- ❑ A ***compatibility table*** shows which operations are compatible.
- ❑ E.g., {Read, Write}

	Read	Write
Read	yes	no
Write	No	no

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

---

ANSI SQL2 Isolation Levels

- SET TRANSACTION READ ONLY | READ WRITE

  [ISOLATION LEVEL READ UNCOMMITTED |
  READ COMMIT |
  REPEATABLE READ |
  SERIALIZABLE]

Concurrency Control Schemes

- Lock-based CC schemes
  - **Two-phase locking** [IBM DB2, SQLServer]
  - Multigranularity locking
  - Tree/Index locking
- **Multiversion** [Oracle, SQLServer]
- Timestamp-based
- Optimistic CC & Certifiers

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: <u>other</u> transactions are allowed to <u>read</u> x

  - **write_lock (x)**
    - x is write-locked by T_i
    - **exclusive** lock: <u>single</u> transaction holds the lock on x
  - **unlock (x)**

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.
     - <u>No locks</u> can be released in this phase
  2. **A Shrinking phase**
     - Once a scheduler has released a lock for a transaction, it <u>cannot request</u> 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); ✓

Rigorous 2PL or industrial Strict 2PL

- The growing phase
  - transactions request locks just before they operate on a data item.
- The growing phase ends at *commit time*.
  - no locks can be released until commit or abort time.
  - no overwriting of dirty data.
  - no overwriting of data read by active transactions.
  - no reading of dirty data.
- Easy to implement a strict 2PL. Why ?
- Has a functional advantage. What ?

Deadlocks

Deadlocks

- Examples:

(I)	2 Items	
T_1	T_2	Comments
rl(x)		granted
	rl(y)	granted
wl(y)		T_1 blocked
	wl(x)	T_2 blocked
		(deadlock)

(II)	1 Item	
T_1	T_2	Comments
rl(x)		granted
	rl(x)	granted
wl(x)		T_1 blocked
	wl(x)	T_2 blocked
		(deadlock)

- Example II involves lock conversion
- The scheduler *restarts* any transaction aborted due to deadlock.

Deadlocks

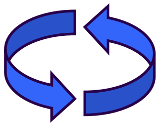
- A *deadlock* occurs when two or more transactions are blocked indefinitely.
- This happens because each holds locks on data items on which the other transaction(s) attempt to place a conflicting lock.
- Necessary conditions for deadlock situations.
  - mutual exclusion
  - hold and wait
  - no preemption
  - circular wait.

Deadlock Handling Schemes

- Deadlock avoidance
  - Timestamp ordering (Wait-Die, Wound-Wait)
- Deadlock Prevention
  - Predeclaration of resources,…
- Deadlock Detection and Resolution
  - Time-out
  - Wait-for graphs

Issues Related to Locking

Deadlock



Livelock

Starvation

Concurrency Control Schemes

- ❑ Lock-based CC schemes
  - **Two-phase locking** [IBM DB2, SQLServer]
  - Multigranularity locking
  - Tree/Index locking
- ❑ **Multiversion** [Oracle, SQLServer]
- ❑ Timestamp-based
- ❑ Optimistic CC & Certifiers

Multiversion Concurrency Control

- ❑ Assume the following sequence of events.

  $W_0(x)\ C_0\ W_2\ (x)\ R_1(x)\ C_2\ C_1$

- ❑ This sequence CANNOT be produced by a strict 2PL scheduler
  - T_1 can not read lock x until after C_2
- ❑ An Idea !!
  - If we had kept the old version of x when $W_2\ (x)$, then we could avoid having to delay T_1 as in 2PL by having T_1 read the previous (old) value of x (produced by T_0).

Basic Idea

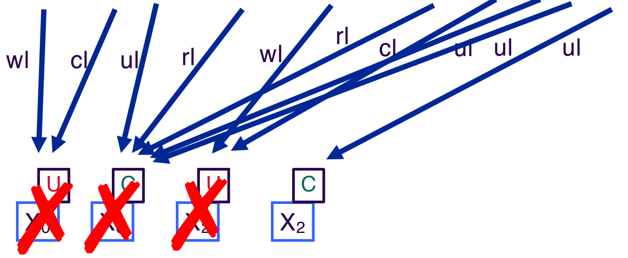
- ❑ The DBMS keeps a list of versions for each x
  - Version x_i means the version of x produced by a Write on x by transaction T_i
- ❑ Each Write(x) produces a new version of x
- ❑ When the scheduler receives a $R_i\ (x)$, it must decide which version of x to read
  - A Read operation will be converted to the form $R_i(x_j)$
- ❑ If a transaction T is aborted, any version it created is destroyed
- ❑ If a transaction T is committed, any version it created becomes available for reading by other transactions

Two Version 2PL (2V2PL)

- keep one or two versions of each data item *x*.
- When a T_i wants to write *x*, it sets a *wl(x)* and creates a new version of *x*, x_i.
  - The *wl(x)* prohibits other transactions from writing *x*.
- Readers are allowed to place a *rl* on their write-locked x or the previous version of *x*.
- When T_i commits, the x_i version of x becomes *x*'s unique version (the previous *x* may now be deleted).
- To delete the previous *x* when T_i commits, we need to know that no other transaction reads *x*.
  - Request a commit lock (cl) which conflict with rl
- Deadlocks are possible and indicate non-CSR execution
  - use any deadlock detection or prevention technique.

Example

- $W_0(x)$ C_0 $R_3(x)$ $W_2(x)$ $R_1(x)$ C_2 C_1 C_3
- $W_0(x_0)$ C_0 $C_0(x_0)$ $R_3(x_0)$ $W_2(x_2)$ $R_1(x_0)$ C_2 C_1 C_3 $C_2(x_2)$



Postgres Isolation Levels

- SET TRANSACTION READ ONLY | READ WRITE
  [ISOLATION LEVEL ~~READ UNCOMMITTED~~ |
  READ COMMIT |
  REPEATABLE READ |
  SERIALIZABLE]
- READ COMMITTED is the *default*
  - Not always the most recent/latest one
  - It cannot see even its own uncommitted updates
- REPEADABLE READS always, Why?
- JDBC: dbcon.TRANSACTION_READ_COMMITTED,
  dbcon.TRANSACTION_SERIALIZABLE