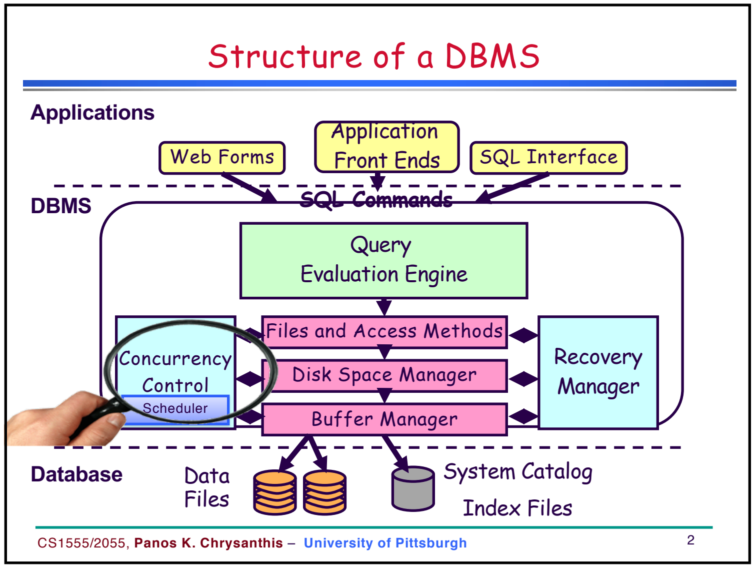
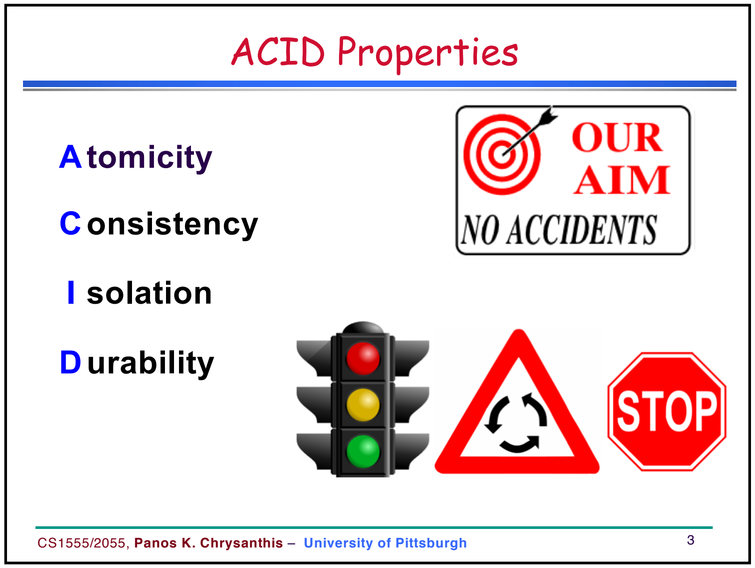
Transaction Processing:
Concurrency control



Structure of a DBMS



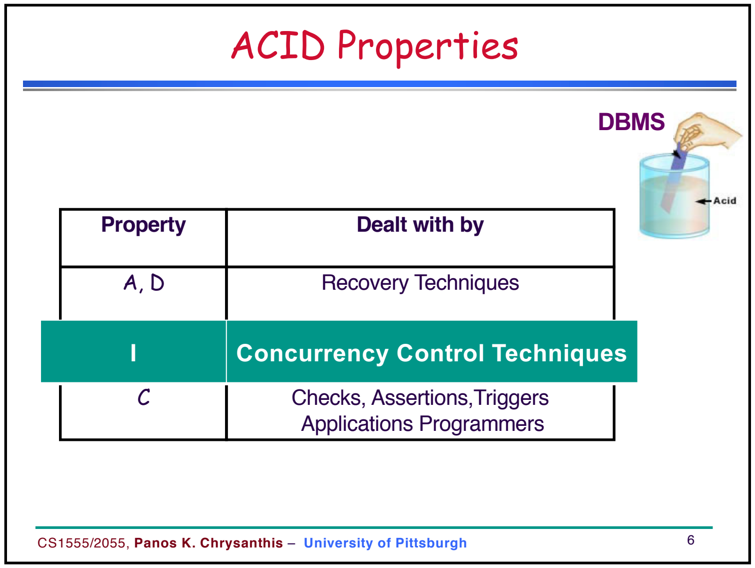
ACID Properties

Atomicity

Consistency

I solation

Durability



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`

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!!

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**

Serial Executions

Item X =100

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

Item X =220

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

Item X =210

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

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 $T_1, T_2, \ldots, T_n$ 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

---

Schedule: example 1

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) commit
Y=Y+N write_item(Y) commit	

Timeline

Mapping:
- Drop local variables, e.g., a, b, c…
- Use db items names, e.g., X and Y, to replace local variables
- Project on the time line

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

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

- Equivalence is defined wrt conflicting operations: the order of any two conflicting operations is the same in S_1 and S_2

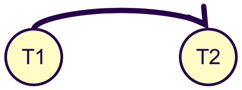
Testing CSR

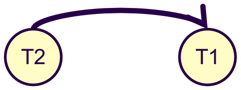
- Test for CSR by analyzing the precedence graph **SG(H)**, called a **serialization graph**, derived from history H.
- An SG(H) is a directed graph in which:
  1. nodes represent transactions in H;
  2. an edge $T_i \rightarrow T_j$, $i \neq j$, means that one of T_i's operations precedes and conflicts with one of T_j's operations in H.
- Serializability Theorem:
  A history H is serializable iff SG(H) is **acyclic**.
- Proof: ?

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)

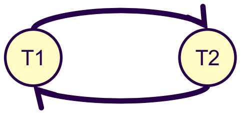
T_1	T_2
	read_item(X)
	X=X+M
	write_item(X)
read_item(X)	
X=X-N	
write_item(X)	
read_item(Y)	
Y=Y+N	
write_item(Y)	





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)	



$T_1 \rightarrow T_2 : w_1(x) <_H w_2(x)$

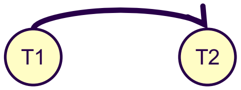
& $r_1(x) <_H w_2(x)$

$T_2 \rightarrow T_1 : r_2(x) <_H w_1(x)$

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

Finding The Equivalent Serial History

- ❑ Make a topological sorting of the serialization graph.

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

SG(H)

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

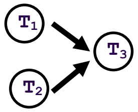
T_2 T_1 T_3

Finding The Equivalent Serial History …

- ❑ Several serial orders can be obtained by topological sorting:

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

SG(H')



Hs: $T_1\ T_2\ T_3$ or $T_2\ T_1\ T_3$

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.

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: 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)**

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); ✓

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 ?

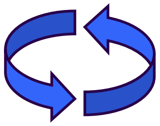
© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com

search ID: vsh0191

"Thank you for holding."

Issues Related to Locking

Deadlock



Livelock

Starvation

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

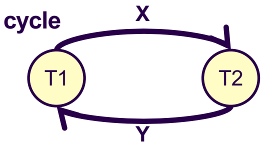
- A *deadlock* occurs when two or more transactions are blocked indefinitely.
- This happens because each holds locks on data items on which the other transaction(s) attempt to place a conflicting lock.
- Necessary conditions for deadlock situations.
  - mutual exclusion
  - hold and wait
  - no preemption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

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?**



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.

  $\boldsymbol{W_0(x)\ C_0\ W_2(x)\ R_1(x)\ C_2\ C_1}$

- 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 $R(x_i)$
- 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 Transaction

- **Class** (classid, max_num_students, cur_num_students)
- Consider transaction “Enroll_student”

```
SET TRANSACTIONS READ WRITE NAME ‘Enroll_student’;
SELECT max_num_students, cur_num_students
FROM    class
WHERE classid = 1

If  (cur_num_students <max_num_students)
    update class
    set cur_num_students = cur_num_students +1
    where classid = 1
else
    print 'the class is full'
COMMIT;
```

Read(classid =1)

Write(classid =1)

Concurrent Transactions

```
SET TRANSACTIONS READ WRITE NAME ‘Enroll_student’;
SELECT max_num_students, cur_num_students
FROM    class
WHERE classid = 1

sleep…

If  (cur_num_students <max_num_students)
    update class
    set cur_num_students = cur_num_students +1
    where classid = 1
else
    print 'the class is full'
COMMIT;
```

Write/Exclusive Lock in Oracle

- Example:

```
SELECT max_num_students,cur_num_students
FROM CLASS
WHERE classID = 1555
FOR UPDATE OF cur_num_students;
```

- Error Messages:
  - No lock: ORA-08177: Cannot serialize access for this transaction”
  - With lock: “The class is full”

Oracle Isolation Levels

- SET TRANSACTION READ ONLY | READ WRITE
  [ISOLATION LEVEL READ COMMITTED| |
  SERIALIZABLE]
- READ COMMITTED is the *default*
  - Not always the most recent/latest one
- REPEADABLE READS always, Why?
- JDBC: dbcon.TRANSACTION_READ_COMMITTED,
  dbcon.TRANSACTION_SERIALIZABLE

Oracle Snapshot Isolation

- Snapshot Isolation, reads come from one instance (snapshot) and write from another.
- Only write conflicts are checked

  if $WS(T_i) \cap WS(T) \neq \emptyset$ **then v**alid = false
  - Oracle: **First Committer Wins**
  - SQLServer, PostgreSQL: **First Updater Wins**
- Snapshot Isolation is weaker than (Conflict) Serializability
- *Write Skew: Assume x and y are related*

  r1(x) r2(x) r1(y) r2(y) w1(x) w2(y)

  x: savings, y: checking, x+y > 0 , w1(x,-75), W2(y,-75)
- *Read Consistency: a read-only transaction T sees the most recent committed DB value at the beginning of T*