

Yannis Papakonstantinou, UCSD

## Integrity and Correctness of Data

- data must be correct at all times
- data must satisfy predicates/constraints
  - some are enforced by data model
    - x is key of relation R
    - attribute A is of type T
  - some are enforced with help by constraints and triggers
    - no employee should make more than twice the average salary
- Two definitions
  - consistent state: a database state that satisfies all constraints
  - consistent DB: a DB in consistent state

1

Yannis Papakonstantinou, UCSD

## Can a DB be Always Consistent ? No, Transactions are the Unit of Consistency

- A DB can *not* be always consistent (it does not even make sense)
  - e.g., when transferring money from your checking to your saving account for a fraction of millisecond your total balance is higher/lower than in reality.
- Transaction: A collection of actions that preserve consistency
- Goal of the database system:
  - given transactions that
    - start with consistent state and end with consistent state when run successfully and in isolation to completion
  - guarantee correctness even if concurrency or failures

2

Yannis Papakonstantinou, UCSD

## In the Next Episodes

- What can violate correctness
  - Failures
  - Concurrency / Data Shring
  - Combinations
- How we can prevent/fix the above violations of correctness
  - Recovery
  - Concurrency control
- Will not consider how to write correct transactions

3

Yannis Papakonstantinou, UCSD

## Failure Model

- System crash with
  - main memory loss
  - cpu halts and resets
- Worse conditions will not be considered
  - disk data lost
  - computer starts nuclear war - nobody survives to reset it

4

Yannis Papakonstantinou, UCSD

## Important Operations of a Transaction

<ul style="list-style-type: none"> <li>• INPUT(X)           <ul style="list-style-type: none"> <li>– retrieve the block of item X from disk</li> </ul> </li> <li>• OUTPUT(X)           <ul style="list-style-type: none"> <li>– force output the block of item X to disk</li> </ul> </li> <li>• READ(X,t)           <ul style="list-style-type: none"> <li>– retrieve item X in variable t. If X is not in memory INPUT(X)</li> </ul> </li> <li>• WRITE(X,t)           <ul style="list-style-type: none"> <li>– write item X. Does not have to be pushed to disk unless buffer manager decides so</li> </ul> </li> <li>• Example: a transaction that doubles A and B           <ul style="list-style-type: none"> <li>– constraint A=B</li> </ul> </li> </ul>	<pre> Read(A,t) ; t:=t*2 Write(A,t) Read(B,t) ; t:=t*2 Write(B,t) Output(A) Output(B)           </pre>
---	--

5

Yannis Papakonstantinou, UCSD

## Key Problem: Unfinished Transactions

- Output A, then the CPU halts before it outputs B
- Database ensures *atomicity*
  - execute all actions of a transaction or none at all

6

Yannis Papakonstantinou, UCSD

## UnDo Logging

Before output on disk write in a log previous value

<i>Memory</i> A: 16 B: 16	<i>Disk</i> A: 16 B: 8	<i>Log</i> <i>TI, Start</i> <i>TI, A, 16</i> <i>TI, B, 8</i> <i>TI, COMMIT</i>
---------------------------------	------------------------------	--

If computer breaks down in this state,  
after it restarts we know to put 16 in Disk

*Two assumptions*

- write operations on log are executed in the order given
- disk/log do not get lost

7

Yannis Papakonstantinou, UCSD

## Undo Logging Rules

- For every write action generate undo log record containing old value
- Before x is modified on disk make sure that log records pertaining to x are in the disk (log)
  - WAL: Write Ahead Logging
- Before COMMIT is flushed to LOG all writes of the transaction must be on disk

8

Yannis Papakonstantinou, UCSD

## Undo Logging Recovery Rules

- Let S = set of transactions T with
  - <T, START> in log
  - **NO** <T, Commit> (or <T, Abort>) in log
- For each <T, X, v> in log in reverse order do
  - if T in S then Write(X,v) ; Output(X)
- For each T in S do
  - write <T, ABORT> to log

*What if failure during recovery ?*

9

Yannis Papakonstantinou, UCSD

## Redo Logging (Deferred Modification)

- Why not Undo logging
- Redo Logging Rules:
  - for every action generate Redo log containing *new* value
  - Before X is modified on disk (DB) all log records for the transaction that modified X must be on disk including the COMMIT record

<i>Memory</i> A: 16 B: 16	<i>Disk</i> A: 8 B: 8	<i>Redo Log</i> <i>TI, Start</i> <i>TI, A, 16</i> <i>TI, B, 16</i> <i>TI, COMMIT</i>
<i>Memory</i> A: 16 B: 16	<i>Disk</i> A: 16 B: 16	<i>Redo Log</i> <i>TI, Start</i> <i>TI, A, 16</i> <i>TI, B, 16</i> <i>TI, COMMIT</i>

10

Yannis Papakonstantinou, UCSD

## Redo Logging Recovery Rules + Checkpoints

- Redo Logging Recovery Rules
  - Let S = set of transactions T with
    - <T, Commit> in log
  - For each <T, X, v> in log in forward order do
    - if T in S then Write(X,v) ; Output(X)
- Recovery is very slow !
  - have to redo all transactions that ever run on the system
- Checkpoints: Periodically do the following
  - Stop accepting transactions and finish all running ones
  - Flush all log records to disk and all buffers to the DB
  - Write CHECKPOINT record on log
- Upon recovery search until the first checkpoint

11

Yannis Papakonstantinou, UCSD

## Buffering Requirements of Undo/Redo

- When the program requests a *write(X)* a new value for X is placed in the main memory (buffer) copy of X
- In general,
  - the program or the buffer manager **can** force X to be output to disk using an *output(X)* command.
  - the program or the buffer manager are **not** required to force X to be output to disk
- However, undo logging and redo logging limit the above possibilities
  - **Redo** logging requires that no item X is stored on disk before the log entries (including the commit entry) are on disk
  - **Undo** requires that all items X written by a transaction are copied back to disk before the commit entry appears in log

12

Yannis Papakonstantinou, UCSD

## Undo/Redo Logging

- Undo/Redo logging does not have the buffer (memory) management restrictions of undo or redo
- Undo/Redo logging rules
  - for every write action generate a log record.  
 $\langle \text{transaction, item, old value, new value} \rangle$
  - (as usual) generate *start*, *commit*, and *abort* records
  - before  $X$  is modified on disk log records pertaining to  $X$  must be on disk
- Undo/Redo recovery
  - first perform Undo recovery, then Redo

13

Yannis Papakonstantinou, UCSD

## Real World Actions

- “Output” commands to other media can not be undone
  - show output to user
  - dispense cash at ATM
- Execute real world actions after commit

```

read(checking, c)
c = c - 100
dispense $100
CRASH
write(checking, c)
      
```

ATM

14

Yannis Papakonstantinou, UCSD

## Concurrency Control

- Multiple transactions run in parallel
- if they run in isolation they are correct
- but what if they run concurrently ?
- when does the *schedule* of execution of some transactions produces a consistent DB ?

Example: Two transactions that maintain the constraint  $A=B$ .

Serial Schedules by definition result in consistent DBs

Serial Schedule A	Serial Schedule B
Read(A);	Read(A)
A:=A+100	A:=2*A
Write(A)	Write(A)
Read(B)	Read(B)
B:=B+100	B:=2*B
Write(B)	Write(B)

15

Yannis Papakonstantinou, UCSD

## Serializable and Non-Serializable Schedules

Bad Schedule	Good Schedule
Read(A);	Read(A);
A:=A+100	A:=A+100
Write(A)	Write(A)
	Read(A)
	A:=2*A
	Write(A)
	Read(B)
	B:=2*B
	Write(B)
Read(B)	Read(B)
B:=B+100	B:=2*B
Write(B)	Write(B)

A simple test of correctness: Can we swap the statements of the schedule so that we produce an equivalent serial schedule ? If yes, the schedule is a serializable one, i.e., a “good” one.

16

Yannis Papakonstantinou, UCSD

## May Swap Non-Conflicting Read/Write Statements

- May swap non *read/write* operations
- May swap *read/write* operations that refer to different items
- May swap two *read(X)*
  - because the same value will be read by both transactions
- Can not swap two *write(X)*
  - the eventual value of  $X$  depends on who makes the last write
- Can not swap *read(X)* with *write(X)*
  - if *read(X)* is moved after *write(X)* it will read a different value for
- Can not swap *write(X)* with *read(X)*

**Good Schedule**

```

Read(A);
A:=A+100
Write(A)
Read(B)
B:=B+100
Write(B)
      
```

**Bad Schedule**

```

Read(A);
A:=A+100
Write(A)
Read(B)
B:=B+100
Write(B)
      
```

Can not swap

17

Yannis Papakonstantinou, UCSD

## Conflict

**Bad Schedule**

```

Read(A);
A:=A+100
Write(A)
Read(A)
A:=2*A
Write(A)
Read(B)
B:=2*B
Write(B)
      
```

Can not swap

18

Yannis Papakonstantinou, UCSD

## Testing for Serializability

- Construct precedence graph  $P(S)$  of schedule  $S$ 
  - Nodes: transactions  $T$  of  $S$
  - Edges:  $T_i \rightarrow T_j$  whenever
    - $p_i(X), q_j(X)$  are actions in  $T_i, T_j$
    - $p_i(X)$  precedes  $q_j(X)$  in  $S$
    - at least one of  $p_i(X), q_j(X)$  is write
- Theorem: A schedule  $S$  is equivalent to a serial schedule  $S'$  if  $P(S) = P(S')$
- Theorem: A schedule  $S$  is serializable if  $P(S)$  is acyclic

Serial Schedule A	Good Schedule
$T_1$ $T_2$	$T_1$ $T_2$
Read(A); A:=A+100 Write(A) Read(B) B:=B+100 Write(B)	Read(A); A:=A+100 Write(A) Read(A) A:=2*A Write(A) Read(B) B:=B+100 Write(B) Read(B) B:=2*B Write(B)

the precedence graph for both is

$T_1 \longrightarrow T_2$

19

Yannis Papakonstantinou, UCSD

## Testing for Serializability (example)

**Bad Schedule**

$T_1$	$T_2$
Read(A); A:=A+100 Write(A)	Read(A) A:=2*A Write(A) Read(B) B:=2*B Write(B)

20

Yannis Papakonstantinou, UCSD

## How to Enforce Serializable Schedules ?

- Scheduler prevents  $P(S)$  cycles from occurring
- Locking operations
  - lock (exclusive):  $l_i(x)$
  - unlock:  $u_i(x)$
- Locking protocol rules
  - Well-formed transactions  $T_1 \dots l_i(x) \dots p_i(x) \dots u_i(x) \dots$
  - Scheduler locking rules  $S \dots l_i(x) \dots u_j(x) \dots$  **no  $l_j(x)$**
  - Two phase (2PL) locking  $T_2 \dots l_i(x) \dots u_i(x) \dots$  **no unlocks no locks**

enhanced with lock operations

if a lock request violates protocol the requesting transaction goes in *wait* state

21

Yannis Papakonstantinou, UCSD

## 2PL is needed for Serializability

Two well-formed transactions on a **non 2PL scheduler**

$T_1$	$T_2$
$l_1(A);$ Read(A); A:=A+100 Write(A); $u_1(A)$	$l_2(A);$ Read(A) A:=2*A Write(A); $u_2(A)$ $l_2(B);$ Read(B) B:=2*B Write(B); $u_2(B)$

**non serializable**

2PL versions of the transactions

$T_1$	$T_2$
$l_1(A);$ Read(A); A:=A+100 Write(A); $l_1(B);$ Read(B) B:=B+100 Write(B); $u_1(A); u_1(B)$	$l_2(A);$ Read(A) A:=2*A Write(A); $l_2(B);$ Read(B) B:=2*B Write(B); $u_2(A); u_2(B)$

**serial(izable)**

22

Yannis Papakonstantinou, UCSD

## 2PL causes deadlocks

- Sequence of lock requests + 2PL requirement may cause deadlock
- One or more of the deadlocked transactions is aborted (undone)
- Aborted deadlocked transactions do not count for the final schedule

$T_1$	$T_2$
$l_1(A);$ Read(A); A:=A+100 Write(A);	$l_2(B);$ Read(B) B:=2*B Write(B);

$l_1(B); T_1$  waits       $l_2(A); T_2$  waits

**DEADLOCK**

23

Yannis Papakonstantinou, UCSD

## The Set of 2PL Schedules is Subset of the Set of Serializable Schedules

We show that the following serializable schedule could not be produced by a Two Phase Lock Scheduler

**Serializable but not 2PL**

$T_1$	$T_2$	$T_3$
Read(A); A:=A+100 Write(A)	Read(A) A:=A+100 Write(A)	Read(B) B:=B+100 Write(B)

impossible to lock(A) because it has to lock(B) because it has to lock(B) later (2PL's rule: no unlock until the last lock

24

Yannis Papakonstantinou, UCSD

## Shared Locks

- Motivation: Allow concurrent read operations on the same object
- Locking operations
  - lock exclusive:  $lx_f(x)$
  - lock shared:  $ls_f(x)$
  - unlock exclusive:  $ux_f(x)$
  - unlock shared:  $us_f(x)$
  - unlock (whatever):  $uf_f(x)$
- Locking protocol
  - Well-formed transactions
    - read between shared locks  $T_1 \dots ls_f(x) \dots r_f(x) \dots uf_f(x) \dots$
    - write/read between exclusive locks  $T_1 \dots lx_f(x) \dots w_f(x)/r_f(x) \dots uf_f(x) \dots$
    - What about transactions that read and write the same object
      - may request exclusive lock in advance
      - upgrade  $T_1 \dots ls_f(x) \dots r_f(x) \dots lx_f(x) \dots w_f(x) \dots lx_f(x) \dots w_f(x)$

25

Yannis Papakonstantinou, UCSD

## Shared Locks (cont'd)

- Locking protocol
  - Well-formed transactions (see previous page)
- Scheduler locking rules
 

	$ls_f(X)$	$lx_f(X)$
$ls_f(X)$	OK	NO
$lx_f(X)$	NO	NO
- 2PL, no change except for upgrades
  - upgrades are allowed in the growing phase

26

Yannis Papakonstantinou, UCSD

## An Alternative Definition of Correctness: View Serializability

- A schedule  $S$  is **view equivalent** to a schedule  $S'$  if
  - every write operation writes the same value
  - every read operation reads the same value
- A schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule  $S'$
- The set of view serializable schedules is a superset of the set of conflict serializable ones
  - schedules with "blind" writes may be view but not conflict serializable
  - blind writes: consecutive writes on the same item by a transaction, without intermediate reads by any other transaction

a schedule that is view but not conflict serializable

27

Yannis Papakonstantinou, UCSD

## Increment Locks

- Atomic Increment Action
  - $in_f(x) : (li_f(x) \dots in_f(x) \dots uf_f(x))$
  - $\dots li_f(x) \dots in_f(x) \dots uf_f(x) \dots$
- $in_f(x), in_g(x)$  do NOT conflict

Swap Increments	$ls_f(X)$	$lx_f(X)$	$li_f(X)$
Read(A)			
Incr(A)			
Incr(A)			
Read(A)			

28

Yannis Papakonstantinou, UCSD

## How does locking work in practice ?

- every system is different
  - may not provide serializability (OLAP, long transactions)
  - may be possible to override serializability
- common simple approach
  - don't trust transaction to request/release locks: system gets them automatically
  - hold all locks until COMMIT
- hash table for locks

29

Yannis Papakonstantinou, UCSD

## Cascading Aborts and Strict 2-Phase Lock (2PL)

- If a transaction  $T$  used a value  $X$  that has been written by a transaction  $T'$  that has (or will be) aborted then  $T$  also has to be aborted
  - and every transaction  $T''$  that used a value written by  $T'$  should also be aborted, and so on recursively
- Strict 2PL is a 2PL version where write locks are not released until the commit.
- Strict 2PL avoids cascading aborts

$T_1$	$T_2$
Read(A): $A := A + 100$ Write(A)	Read(A) $A := 2 * A$ Write(A)
Read(B) $B := B + 100$ Write(B)	Read(B) $B := 2 * B$ Write(B)

if transaction  $T_1$  aborts then  $T_2$  must also abort

this schedule can not be produced by a strict 2PL but can be produced by 2PL

30

## Granularity of locks

- what are the locked items ?
  - relations ? tuples ? pages ? attributes ?
- if we lock large objects
  - need few locks
  - low concurrency
- if we lock small objects
  - need more locks
  - higher concurrency