

# CS150A Database

Lu Sun

School of Information Science and Technology

ShanghaiTech University

Nov. 1, 2022

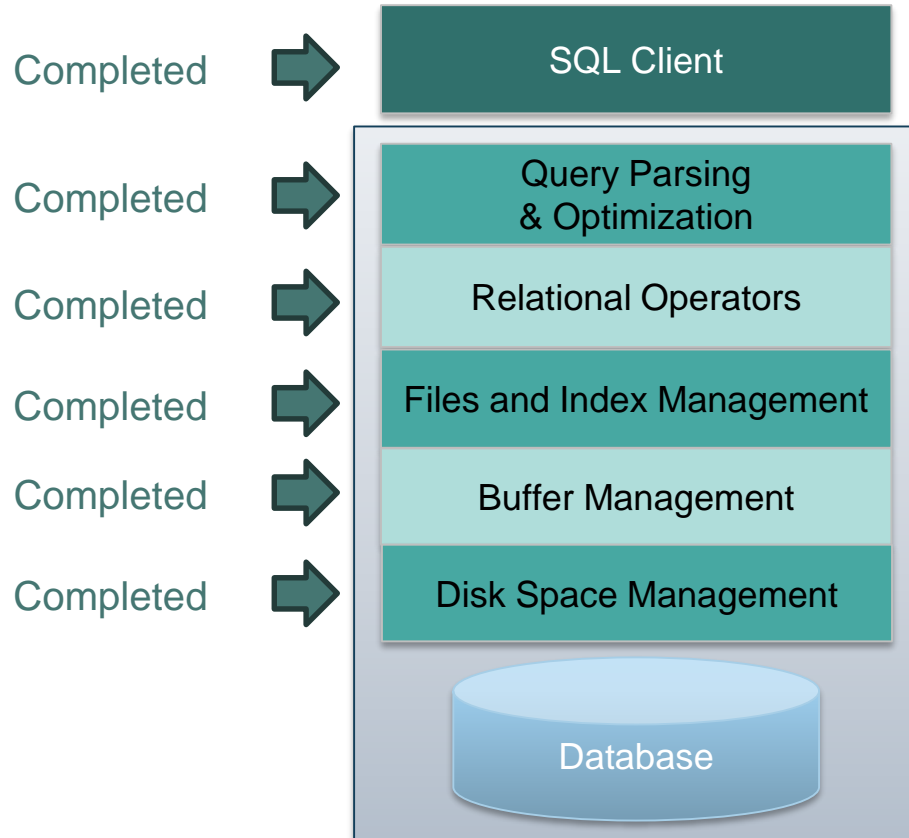
Today:

- Transactions & Concurrency  
Control I:

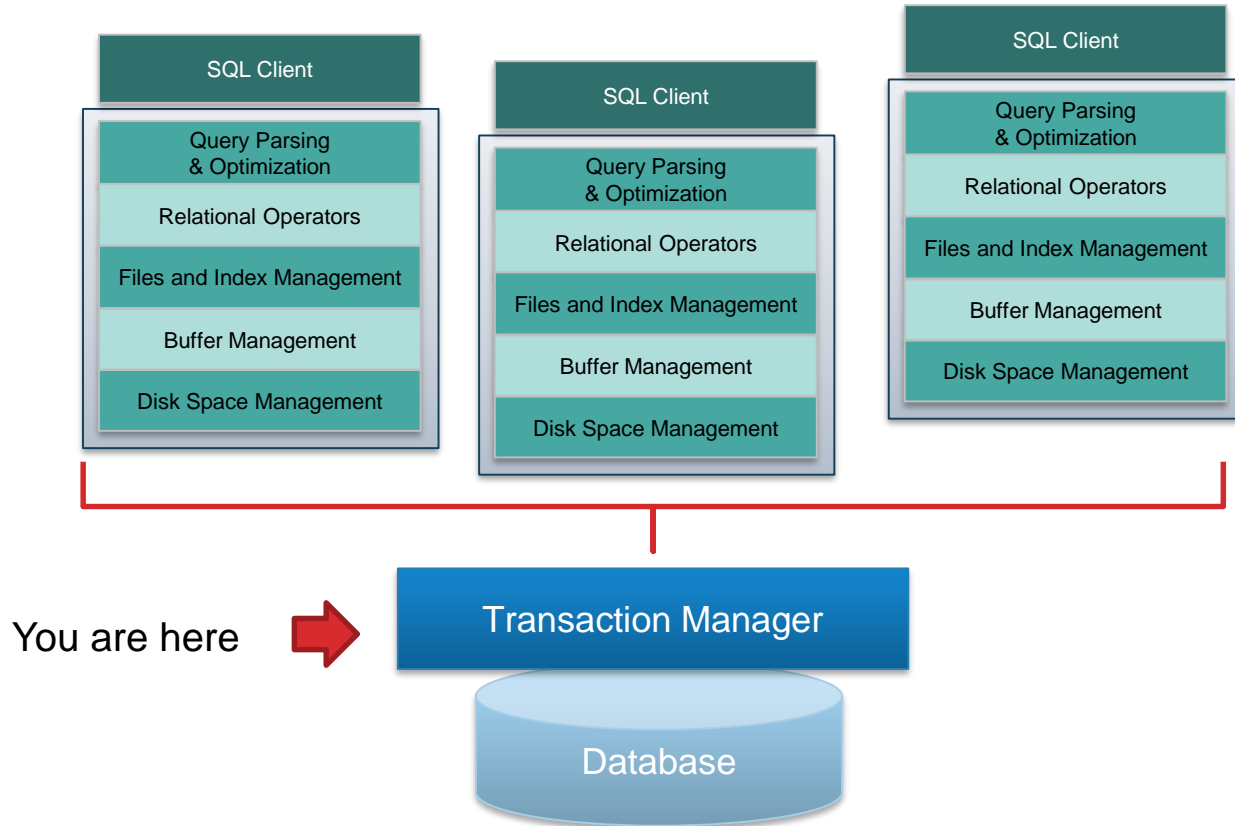
Readings:

- Database Management Systems  
(DBMS), Chapters 16&17

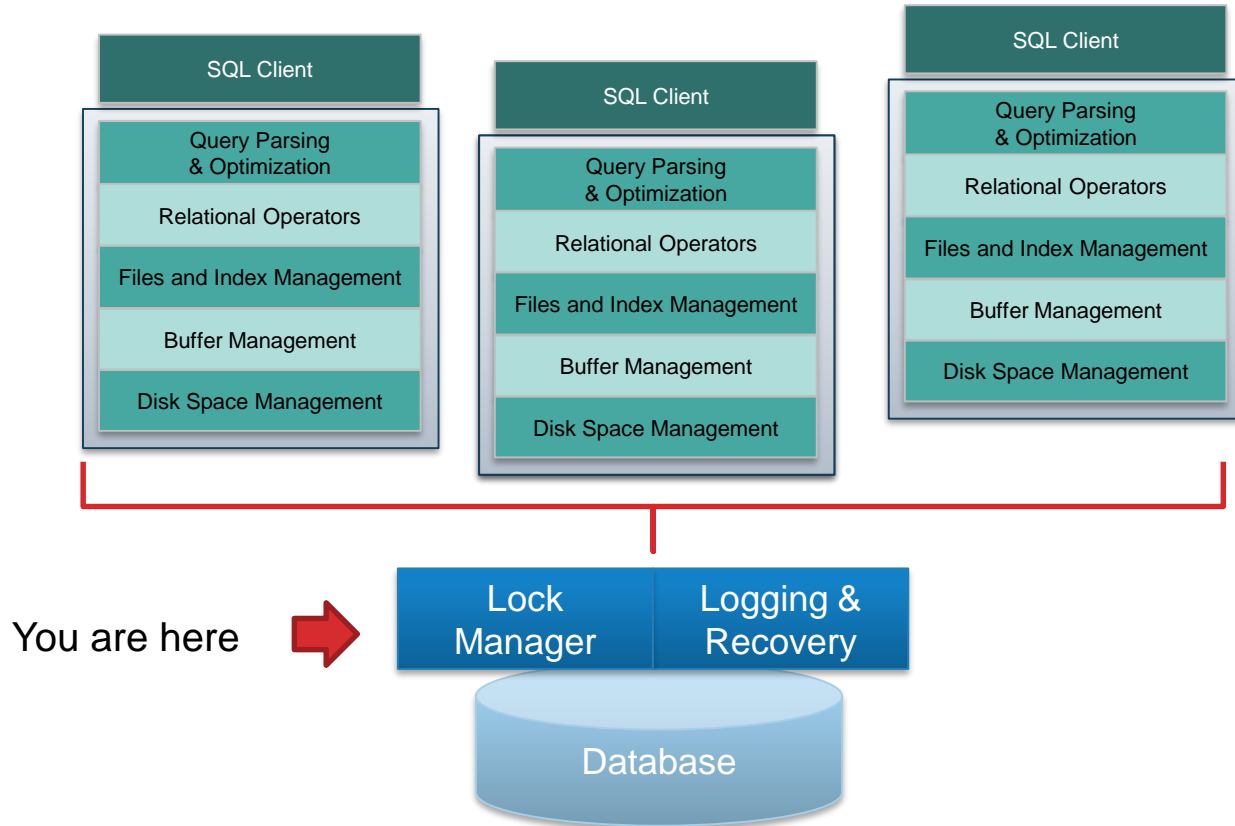
# Architecture of a DBMS



# Architecture of a DBMS, Part 2



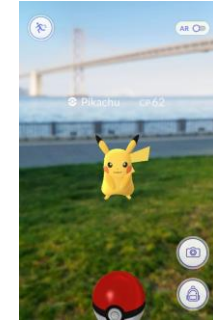
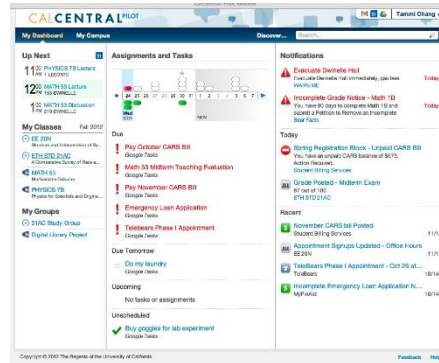
# Architecture of a DBMS, Part 3



# Applications on DBMS

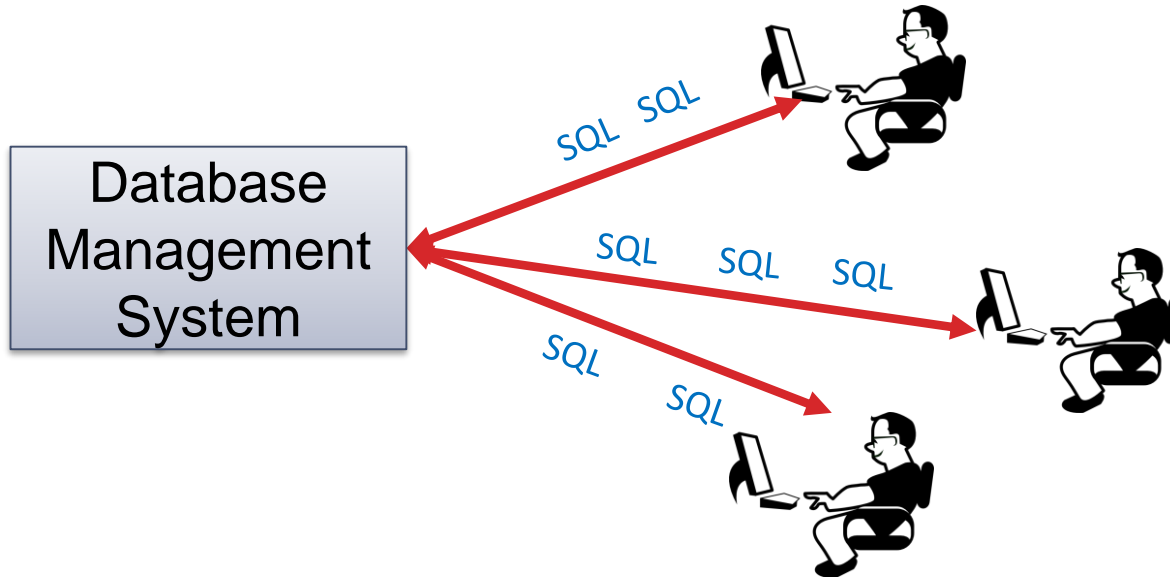
- Virtually any compute service that maintains state today is an application on top of some kind of DBMS

- Uber
- Kayak
- Amazon.com
- BankofAmerica
- Pokemon Go



# Applications Want Something from the DBMS

- Queries and updates of course: what you learned so far!
- Real applications are composed of many statements being generated by user behaviors
- Many users work with the application at the same time



# Concurrency Control & Recovery

- **Part 1: Concurrency Control**
  - Correct/fast data access in the presence of concurrent work by many users
  - Disorderly processing that provides the illusion of order
- **Part 2: Recovery**
  - Ensure database is fault tolerant
  - Not corrupted by software, system or media failure
  - Storage guarantees for mission-critical data
- **It's all about the programmer!**
  - Systems provide guarantees
  - These guarantees lighten the load of app writers

# Concurrent Execution: Why bother?

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are twofold:
  - *Throughput* (transactions per second):
    - Increase processor/disk utilization → more transactions per second (TPS) completed
      - Single core: can use the CPU while another xact is reading to/writing from the disk
      - Multicore: ideally, scale throughput in the number of processors
  - *Latency* (response time per transaction):
    - Multiple transactions can run at the same time
    - So one transaction's latency need not be dependent on another unrelated transaction
    - Or that's the hope
- Both are important!



# Motivating Example

```
UPDATE Budget  
SET money = money - 500  
WHERE pid = 1
```

```
UPDATE Budget  
SET money = money + 200  
WHERE pid = 2
```

```
UPDATE Budget  
SET money = money + 300  
WHERE pid = 3
```

```
SELECT sum(money)  
FROM Budget
```

Two Issues:

1. Order matters!
2. Users need a way to say what's OK

# Different Types of Problems

## User 1

```
INSERT INTO DollarProducts(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99

DELETE Product
WHERE price <= 0.99
```

## User 2

```
SELECT count(*)
FROM Product

SELECT count(*)
FROM DollarProducts
```

What could go wrong?

**Inconsistent Reads**

# Different Types of Problems, Part 2

## User 1

```
UPDATE Product  
SET Price = Price - 10.99  
WHERE pname = "CoolToy"
```

## User 2

```
UPDATE Product  
SET Price = Price*0.6  
WHERE pname = "CoolToy"
```

What could go wrong?

**Lost Update**

# Different Types of Problems, Part 3

**User 1**

```
UPDATE Account  
SET amount = 1000000  
WHERE number = "my-account"
```

Aborted by  
the system

**User 2**

```
SELECT amount  
FROM Account  
WHERE number = "my-account"
```

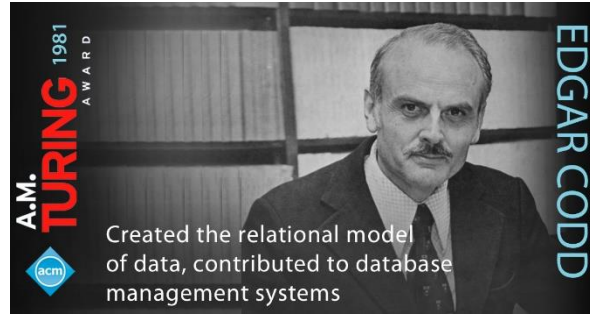
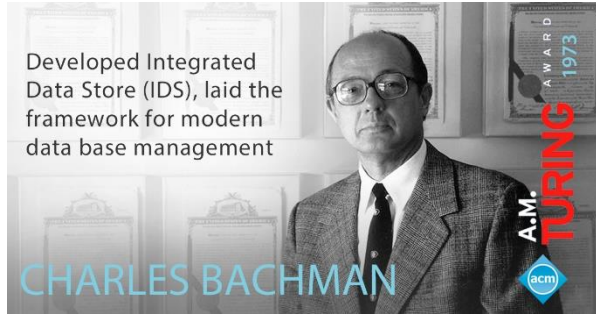
What could go wrong? **Dirty Reads**

# TRANSACTIONS

# Transaction: Concept and Implementation

- Major component of database systems
- Critical for most applications; arguably more so than SQL

# An Aside: Database Turing Awards



# What is a Transaction?

- A sequence of *multiple actions* to be executed as an *atomic* unit
- Application View (SQL View):
  - Begin transaction
  - Sequence of SQL statements
  - End transaction
- Examples
  - Transfer money between accounts
  - Book a flight, a hotel and a car together on Expedia



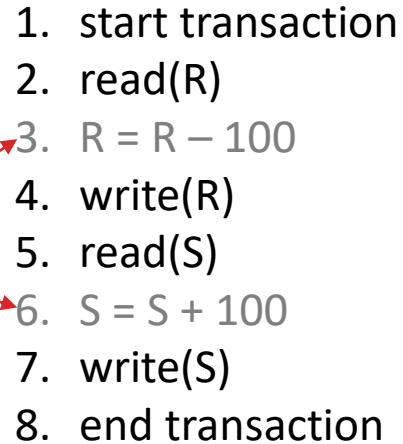
# Our Transaction Model

- **Transaction (“Xact”):**
  - DBMS’s abstract view of an application program (or activity)
    - A sequence of *reads* and *writes* of database objects
    - Batch of work that must *commit* or *abort* as an atomic unit
- **Xact Manager controls execution of transactions**
- **Program logic is invisible to DBMS!**
  - Arbitrary computation possible on data fetched from the DB
  - The DBMS only sees data read/written from/to the DB
  - (Note: modern systems have started rethinking this assumption, but we’ll stick with it here)

# Transaction Example

- Transaction to transfer \$100 from account R to account S

Not seen by the  
DBMS transaction  
manager!

- 
1. start transaction
  2. read(R)
  3.  $R = R - 100$
  4. write(R)
  5. read(S)
  6.  $S = S + 100$
  7. write(S)
  8. end transaction

# ACID: High-Level Properties of Transactions

- **A tomicity:** *All* actions in the Xact happen, or *none* happen.
- **C onsistency:** If the DB *starts* out *consistent*, it *ends* up *consistent* at the end of the Xact
- **I solation:** Execution of *each* Xact is *isolated from* that of *others*
- **D urability:** If a Xact *commits*, its effects *persist*.

Note: This is a mnemonic, not a formalism. We'll do some formalisms shortly.

# I Isolation (Concurrency)

- DBMS interleaves actions of many xacts
  - Actions = reads/writes of DB objects
- DBMS ensures 2 xacts do not “interfere”
- Each xact executes as if it ran by itself.
  - Concurrent accesses have no effect on xact’s behavior
  - Net effect must be identical to executing all transactions in some serial order
  - Users & programmers think about transactions in isolation
    - Without considering effects of other concurrent Xacts!

# Isolation: An Example

- Think about avoiding problems due to concurrency
  - If another transaction T2 accesses R and S between steps 4 and 5 of T1, it will see a lower value for R+S.

T1

1. start transaction
2. read(R)
3.  $R = R - 100$
4. write(R)
5. read(S)
6.  $S = S + 100$
7. write(S)
8. end transaction

T2

1. start transaction
2. read(R)
3. print(R+S)
4. end transaction

- Isolation easy to achieve by running one Xact at a time
  - However, recall that serial execution is not desirable

# Atomicity and Durability

- **A transaction ends in one of two ways:**
  - **Commit** after completing all its actions
    - “commit” is a contract with the caller of the DB
  - **Abort** (or be aborted by the DBMS) after executing some actions
    - Or **system crash** while the xact is in progress; treat as abort.
- **Two key properties** for a transaction
  - **Atomicity:** Either execute all its actions, or none of them
  - **Durability:** The effects of a committed xact must survive failures.
- DBMS typically ensures the above by **logging** all actions:
  - **Undo** the actions of aborted/failed transactions.
  - **Redo** actions of committed transactions not yet propagated to disk when system crashes

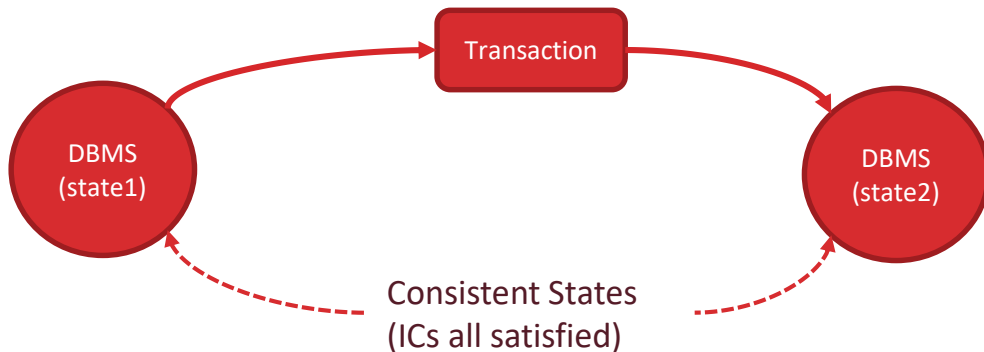
# Atomicity and Durability, cont.

- Atomicity
  - If the transaction fails after step 4 and before step 7
    - Money will be “lost” → inconsistent database
  - DBMS should ensure that updates of a partially executed transaction are not reflected
- Durability
  - Once the user hears that the transaction is complete, can rest easy that the \$100M was transferred from R to S.

1. start transaction
2. read(R)
3.  $R = R - 100$
4. write(R)
5. read(S)
6.  $S = S + 100$
7. write(S)
8. end transaction

# Transaction **C**onsistency

- **Transactions preserve DB consistency**
  - Given a consistent DB state, produce another consistent DB state
- DB consistency expressed as a set of **declarative integrity constraints**
  - CREATE TABLE/ASSERTION statements
- **Transactions that violate integrity are aborted**
  - That's all the DBMS can automatically check!





# Summary

- We have seen an overview
- ACID Transactions make guarantees that
  - Improve performance (via concurrency)
  - Relieve programmers of correctness concerns
    - Hide concurrency and failure handling!
- Two key issues to consider, and mechanisms
  - Concurrency control (via two-phase locking)
  - Recovery (via write-ahead logging WAL)
- We'll do concurrency control first

# CONCURRENCY CONTROL

# Concurrency Control: Providing Isolation

- **Naïve approach - serial execution**
  - One transaction runs at a time
  - Safe but slow
- **Execution must be interleaved for better performance**
- With concurrent executions, how does one **define** and **ensure** correctness?

# Transaction Schedules

T1	T2
begin	
read(A)	
write(A)	
read(B)	
write(B)	
commit	
	begin
	read(A)
	write(A)
	read(B)
	write(B)
	commit

A **schedule** is a sequence of actions on data from one or more transactions.

Actions: Begin, Read, Write, Commit and Abort.

$R_1(A) W_1(A) R_1(B) W_1(B) R_2(A) W_2(A) R_2(B) W_2(B)$

By convention we only include committed transactions, and omit Begin and Commit.

# Serial Equivalence

- We need a “touchstone” concept for correct behavior

- Definition: Serial schedule**

- Each transaction runs from start to finish without any intervening actions from other transactions

- Definition: 2 schedules are equivalent** if they:

- involve the same transactions
- each individual transaction's actions are ordered the same
- both schedules leave the DB in the same final state

S1	S2
T1: r1	
T2: w1	
T3: r2	
T4: w2	
T5: r3	
T6: w3	
T7: r4	
T8: w4	
T9: r5	
T10: w5	
T11: r6	
T12: w6	
T13: r7	
T14: w7	
T15: r8	
T16: w8	
T17: r9	
T18: w9	
T19: r10	
T20: w10	
T21: r11	
T22: w11	
T23: r12	
T24: w12	
T25: r13	
T26: w13	
T27: r14	
T28: w14	
T29: r15	
T30: w15	
T31: r16	
T32: w16	
T33: r17	
T34: w17	
T35: r18	
T36: w18	
T37: r19	
T38: w19	
T39: r20	
T40: w20	
T41: r21	
T42: w21	
T43: r22	
T44: w22	
T45: r23	
T46: w23	
T47: r24	
T48: w24	
T49: r25	
T50: w25	
T51: r26	
T52: w26	
T53: r27	
T54: w27	
T55: r28	
T56: w28	
T57: r29	
T58: w29	
T59: r30	
T60: w30	
T61: r31	
T62: w31	
T63: r32	
T64: w32	
T65: r33	
T66: w33	
T67: r34	
T68: w34	
T69: r35	
T70: w35	
T71: r36	
T72: w36	
T73: r37	
T74: w37	
T75: r38	
T76: w38	
T77: r39	
T78: w39	
T79: r40	
T80: w40	
T81: r41	
T82: w41	
T83: r42	
T84: w42	
T85: r43	
T86: w43	
T87: r44	
T88: w44	
T89: r45	
T90: w45	
T91: r46	
T92: w46	
T93: r47	
T94: w47	
T95: r48	
T96: w48	
T97: r49	
T98: w49	
T99: r50	
T100: w50	
T101: r51	
T102: w51	
T103: r52	
T104: w52	
T105: r53	
T106: w53	
T107: r54	
T108: w54	
T109: r55	
T110: w55	
T111: r56	
T112: w56	
T113: r57	
T114: w57	
T115: r58	
T116: w58	
T117: r59	
T118: w59	
T119: r60	
T120: w60	
T121: r61	
T122: w61	
T123: r62	
T124: w62	
T125: r63	
T126: w63	
T127: r64	
T128: w64	
T129: r65	
T130: w65	
T131: r66	
T132: w66	
T133: r67	
T134: w67	
T135: r68	
T136: w68	
T137: r69	
T138: w69	
T139: r70	
T140: w70	
T141: r71	
T142: w71	
T143: r72	
T144: w72	
T145: r73	
T146: w73	
T147: r74	
T148: w74	
T149: r75	
T150: w75	
T151: r76	
T152: w76	
T153: r77	
T154: w77	
T155: r78	
T156: w78	
T157: r79	
T158: w79	
T159: r80	
T160: w80	
T161: r81	
T162: w81	
T163: r82	
T164: w82	
T165: r83	
T166: w83	
T167: r84	
T168: w84	
T169: r85	
T170: w85	
T171: r86	
T172: w86	
T173: r87	
T174: w87	
T175: r88	
T176: w88	
T177: r89	
T178: w89	
T179: r90	
T180: w90	
T181: r91	
T182: w91	
T183: r92	
T184: w92	
T185: r93	
T186: w93	
T187: r94	
T188: w94	
T189: r95	
T190: w95	
T191: r96	
T192: w96	
T193: r97	
T194: w97	
T195: r98	
T196: w98	
T197: r99	
T198: w99	
T199: r100	
T200: w100	
T201: r101	
T202: w101	
T203: r102	
T204: w102	
T205: r103	
T206: w103	
T207: r104	
T208: w104	
T209: r105	
T210: w105	
T211: r106	
T212: w106	
T213: r107	
T214: w107	
T215: r108	
T216: w108	
T217: r109	
T218: w109	
T219: r110	
T220: w110	
T221: r111	
T222: w111	
T223: r112	
T224: w112	
T225: r113	
T226: w113	
T227: r114	
T228: w114	
T229: r115	
T230: w115	
T231: r116	
T232: w116	
T233: r117	
T234: w117	
T235: r118	
T236: w118	
T237: r119	
T238: w119	
T239: r120	
T240: w120	
T241: r121	
T242: w121	
T243: r122	
T244: w122	
T245: r123	
T246: w123	
T247: r124	
T248: w124	
T249: r125	
T250: w125	
T251: r126	
T252: w126	
T253: r127	
T254: w127	
T255: r128	
T256: w128	
T257: r129	
T258: w129	
T259: r130	
T260: w130	
T261: r131	
T262: w131	
T263: r132	
T264: w132	
T265: r133	
T266: w133	
T267: r134	
T268: w134	
T269: r135	
T270: w135	
T271: r136	
T272: w136	
T273: r137	
T274: w137	
T275: r138	
T276: w138	
T277: r139	
T278: w139	
T279: r140	
T280: w140	
T281: r141	
T282: w141	
T283: r142	
T284: w142	
T285: r143	
T286: w143	
T287: r144	
T288: w144	
T289: r145	
T290: w145	
T291: r146	
T292: w146	
T293: r147	
T294: w147	
T295: r148	
T296: w148	
T297: r149	
T298: w149	
T299: r150	
T300: w150	
T301: r151	
T302: w151	
T303: r152	
T304: w152	
T305: r153	
T306: w153	
T307: r154	
T308: w154	
T309: r155	
T310: w155	
T311: r156	
T312: w156	
T313: r157	
T314: w157	
T315: r158	
T316: w158	
T317: r159	
T318: w159	
T319: r160	
T320: w160	
T321: r161	
T322: w161	
T323: r162	
T324: w162	
T325: r163	
T326: w163	
T327: r164	
T328: w164	
T329: r165	
T330: w165	
T331: r166	
T332: w166	
T333: r167	
T334: w167	
T335: r168	
T336: w168	
T337: r169	
T338: w169	
T339: r170	
T340: w170	
T341: r171	
T342: w171	
T343: r172	
T344: w172	
T345: r173	
T346: w173	
T347: r174	
T348: w174	
T349: r175	
T350: w175	
T351: r176	
T352: w176	
T353: r177	
T354: w177	
T355: r178	
T356: w178	
T357: r179	
T358: w179	
T359: r180	
T360: w180	
T361: r181	
T362: w181	
T363: r182	
T364: w182	
T365: r183	
T366: w183	
T367: r184	
T368: w184	
T369: r185	
T370: w185	
T371: r186	
T372: w186	
T373: r187	
T374: w187	
T375: r188	
T376: w188	
T377: r189	
T378: w189	
T379: r190	
T380: w190	
T381: r191	
T382: w191	
T383: r192	
T384: w192	
T385: r193	
T386: w193	
T387: r194	
T388: w194	
T389: r195	
T390: w195	
T391: r196	
T392: w196	
T393: r197	
T394: w197	
T395: r198	
T396: w198	
T397: r199	
T398: w199	
T399: r200	
T400: w200	
T401: r201	
T402: w201	
T403: r202	
T404: w202	
T405: r203	
T406: w203	
T407: r204	
T408: w204	
T409: r205	
T410: w205	
T411: r206	
T412: w206	
T413: r207	
T414: w207	
T415: r208	
T416: w208	
T417: r209	
T418: w209	
T419: r210	
T420: w210	
T421: r211	
T422: w211	
T423: r212	
T424: w212	
T425: r213	
T426: w213	
T427: r214	
T428: w214	
T429: r215	
T430: w215	
T431: r216	
T432: w216	
T433: r217	
T434: w217	
T435: r218	
T436: w218	
T437: r219	
T438: w219	
T439: r220	
T440: w220	
T441: r221	
T442: w221	
T443: r222	
T444: w222	
T445: r223	
T446: w223	
T447: r224	
T448: w224	
T449: r225	
T450: w225	
T451: r226	
T452: w226	
T453: r227	
T454: w227	
T455: r228	
T456: w228	
T457: r229	
T458: w229	
T459: r230	
T460: w230	
T461: r231	
T462: w231	
T463: r232	
T464: w232	
T465: r233	
T466: w233	
T467: r234	
T468: w234	
T469: r235	
T470: w235	
T471: r236	
T472: w236	
T473: r237	
T474: w237	
T475: r238	
T476: w238	
T477: r239	
T478: w239	
T479: r240	
T480: w240	
T481: r241	
T482: w241	
T483: r242	
T484: w242	
T485: r243	
T486: w243	
T487: r244	
T488: w244	
T489: r245	
T490: w245	
T491: r246	
T492: w246	
T493: r247	
T494: w247	
T495: r248	
T496: w248	
T497: r249	
T498: w249	
T499: r250	
T500: w250	
T501: r251	
T502: w251	
T503: r252	
T504: w252	
T505: r253	
T506: w253	
T507: r254	
T508: w254	
T509: r255	
T510: w255	
T511: r256	
T512: w256	
T513: r257	
T514: w257	
T515: r258	
T516: w258	
T517: r259	
T518: w259	
T519: r260	
T520: w260	
T521: r261	
T522: w261	
T523: r262	
T524: w262	
T525: r263	
T526: w263	
T527: r264	
T528: w264	
T529: r265	
T530: w265	
T531: r266	
T532: w266	
T533: r267	
T534: w267	
T535: r268	
T536: w268	
T537: r269	
T538: w269	
T539: r270	
T540: w270	
T541: r271	
T542: w271	
T543: r272	
T544: w272	
T545: r273	
T546: w273	
T547: r274	
T548: w274	
T549: r275	
T550: w275	
T551: r276	
T552: w276	
T553: r277	
T554: w277	
T555: r278	
T556: w278	
T557: r279	
T558: w279	
T559: r280	
T560: w280	
T561: r281	
T562: w281	
T563: r282	
T564: w282	
T565: r283	
T566: w283	
T567: r284	
T568: w284	
T569: r285	
T570: w285	
T571: r286	
T572: w286	
T573: r287	
T574: w287	
T575: r288	
T576: w288	
T577: r289	
T578: w289	
T579: r290	
T580: w290	
T581: r291	
T582: w291	
T583: r292	
T584: w292	
T585: r293	
T586: w293	
T587: r294	
T588: w294	
T589: r295	
T590: w295	
T591: r296	
T592: w296	
T593: r297	
T594: w297	
T595: r298	
T596: w298	
T597: r299	
T598: w299	
T599: r300	
T600: w300	
T601: r301	
T602: w301	
T603: r302	
T604: w302	
T60	

# Serializability

- **Definition:** Schedule S is **serializable** if:
  - S is equivalent to some serial schedule



# Schedule 1

T1: Transfer \$100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit

- Let T1 transfer \$100 from A to B
- Let T2 add 10% interest to A & B
- Serial schedule in which T1 is followed by T2
  - Final outcome:
    - $A := 1.1 * (A - 100)$
    - $B := 1.1 * (B + 100)$

# Schedule 2

T1: Transfer \$100 from A to B	T2: Add 10% interest to A & B
	begin
	read(A)
	$A = A * 1.1$
	write(A)
	read(B)
	$B = B * 1.1$
	write(B)
	commit
begin	
read(A)	
$A = A - 100$	
write(A)	
read(B)	
$B = B + 100$	
write(B)	
commit	

- Serial schedule in which T2 is followed by T1
  - Final outcome:
    - $A := (1.1 * A) - 100$
    - $B := (1.1 * B) + 100$
  - Different!
    - But still understandable



# Schedule 3

T1: Transfer \$100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
	begin
	read(A)
	A = A * 1.1
	write(A)
read(B)	
B = B + 100	
write(B)	
commit	
	read(B)
	B = B * 1.1
	write(B)
	commit

- Schedule in which actions of T1 and T2 are interleaved.
- This is not a serial schedule
- But it is equivalent to schedule 1
  - $A := (A-100)*1.1$
  - $B := (B+100)*1.1$
- Hence **serializable!**

# Conflicting Operations

- Tricky to check property “**leaves the DB in the same final state**”
- Need an easier equivalence test!
  - Settle for a “conservative” test: always true positives, but some false negatives
  - I.e. sacrifice some concurrency for easier correctness check
- **Use notion of “conflicting” operations (read/write)**
- **Definition: Two operations conflict if they:**
  - Are by different transactions,
  - Are on the same object,
  - At least one of them is a write.
- The order of non-conflicting operations has no effect on the final state of the database!
  - Focus our attention on the order of conflicting operations

# Conflict Serializable Schedules

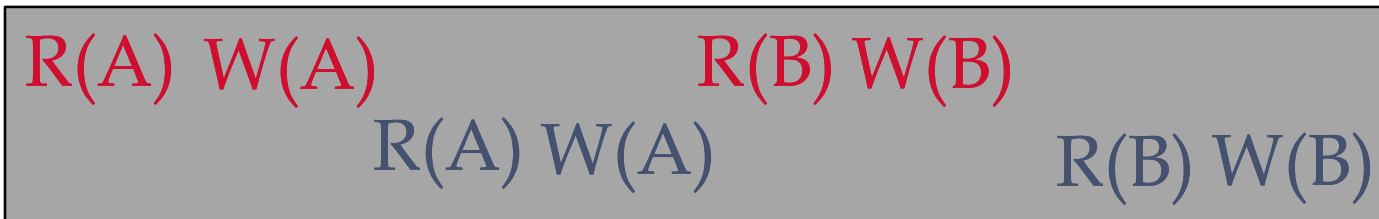
- **Definition: Two schedules are *conflict equivalent* if:**
  - They involve the same actions of the same transactions, and
  - Every pair of conflicting actions is ordered the same way
- **Definition: Schedule S is *conflict serializable* if:**
  - S is conflict equivalent to some serial schedule
  - Implies S is also Serializable

**Note:** some serializable schedules are NOT conflict serializable

- Conflict serializability gives false negatives as a test for serializability!
- The cost of a conservative test
- A price we pay to achieve efficient enforcement

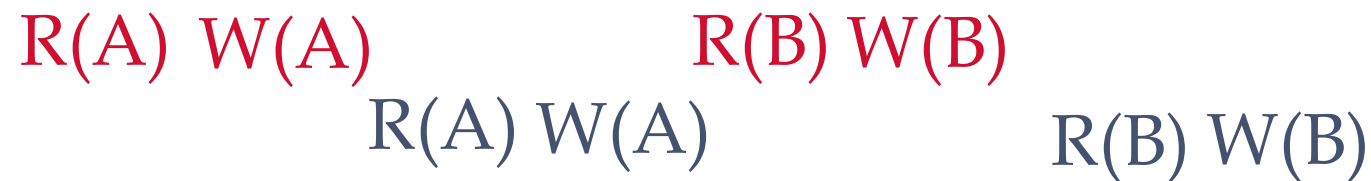
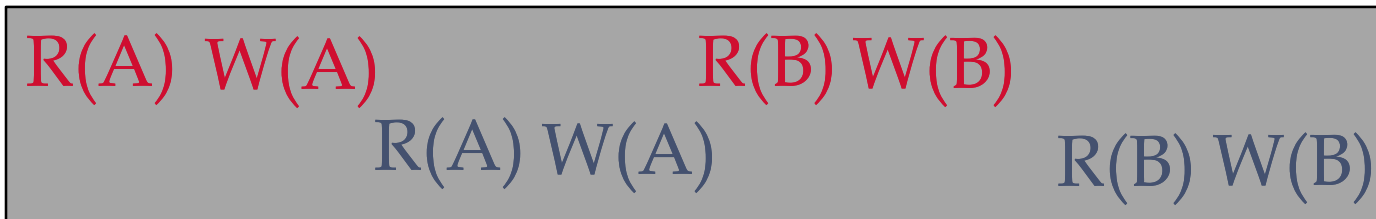
# Conflict Serializability - Intuition

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- ***Example***



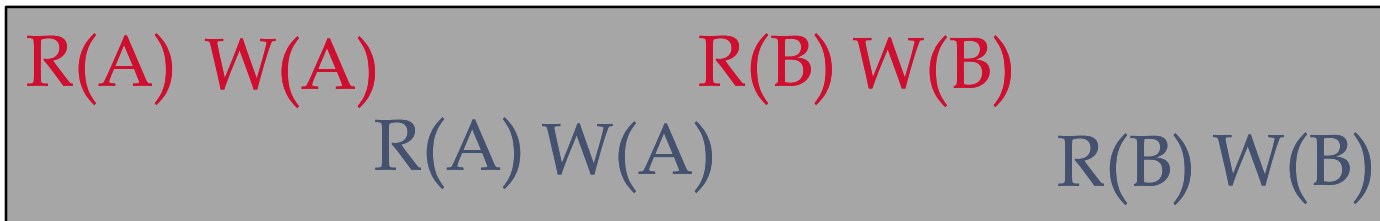
# Conflict Serializability – Intuition, Part 2

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- ***Example***



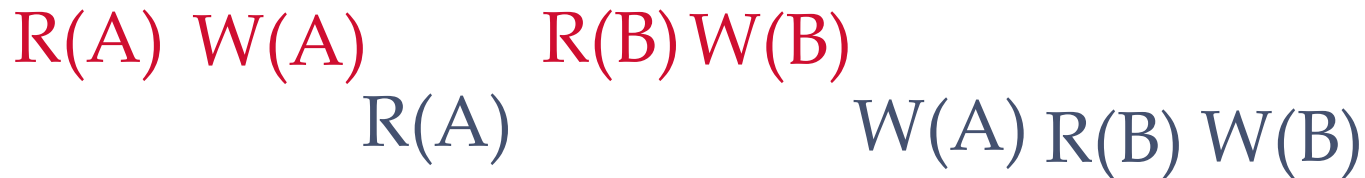
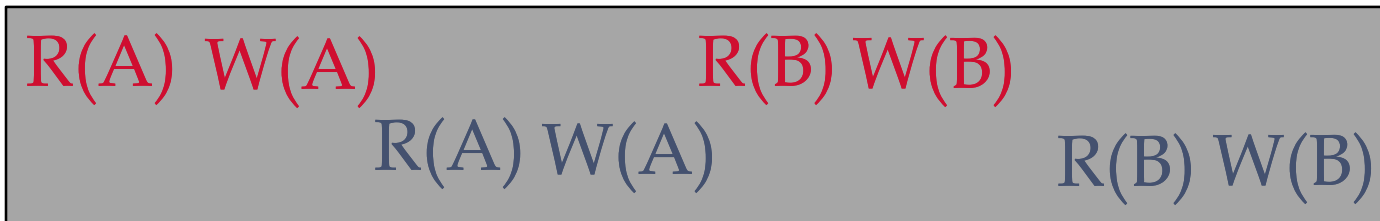
# Conflict Serializability – Intuition, Part 3

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- ***Example***



# Conflict Serializability – Intuition, Part 4

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- **Example**



# Conflict Serializability – Intuition, Part 5

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- **Example**

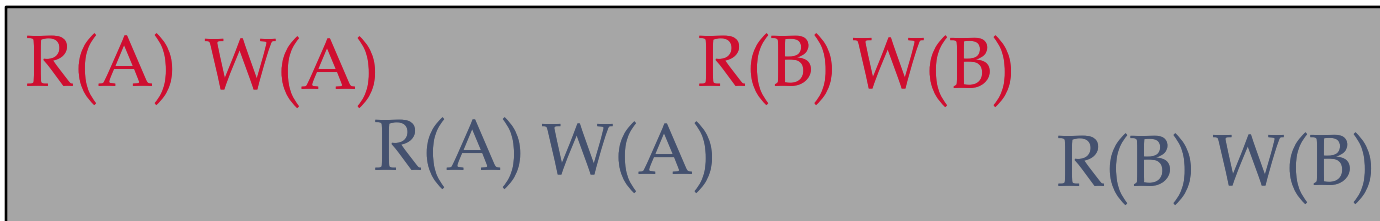


Diagram illustrating a serial schedule S' obtained by swapping consecutive non-conflicting operations of different transactions:

Red (T1): R(A) W(A)

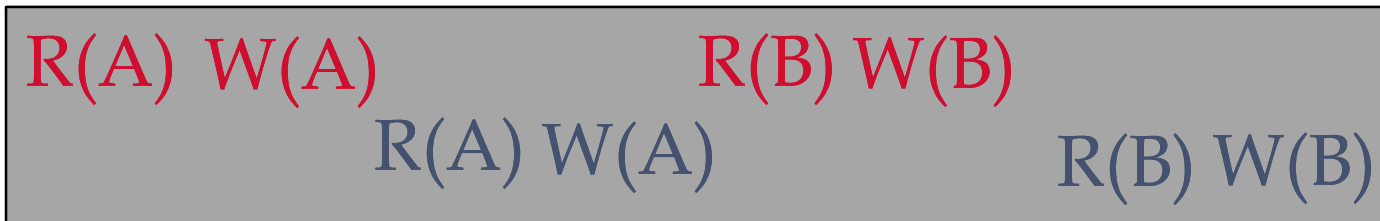
Blue (T2): R(B) W(B)

The serial schedule S' is: R(A) W(A) R(B) W(B).



# Conflict Serializability – Intuition, cont

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- *Example*



R(A) W(A) R(B) W(B)  
R(A)W(A) R(B) W(B)

# Conflict Serializability (Continued)

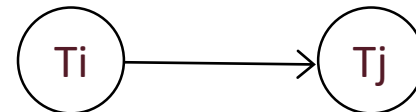
- Here's another example:

$R(A)$   $W(A)$   
 $R(A)$   $W(A)$

- Conflict Serializable or not?

**NOT!**

# Conflict Dependency Graph



- **Dependency Graph:**
  - One node per Xact
  - Edge from  $T_i$  to  $T_j$  if:
    - An operation  $O_i$  of  $T_i$  conflicts with an operation  $O_j$  of  $T_j$  and
    - $O_i$  appears earlier in the schedule than  $O_j$
- **Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic.**

Proof Sketch: Conflicting operations prevent us from “swapping” operations into a serial schedule

# Example

- **A schedule that is not conflict serializable**

T1:        R(A), W(A)
-----------------------

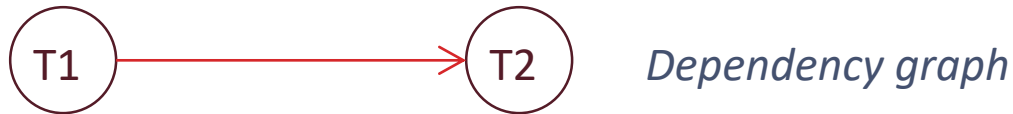


*Dependency graph*

# Example, pt 2

- **A schedule that is not conflict serializable**

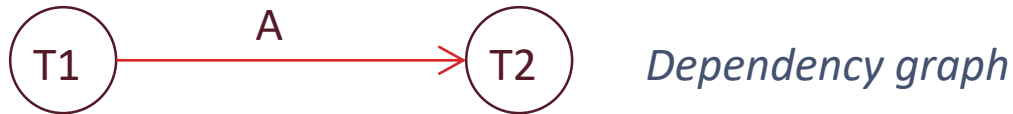
T1:	R(A), W(A),
T2:	R(A)



# Example, pt 3

- **A schedule that is not conflict serializable**

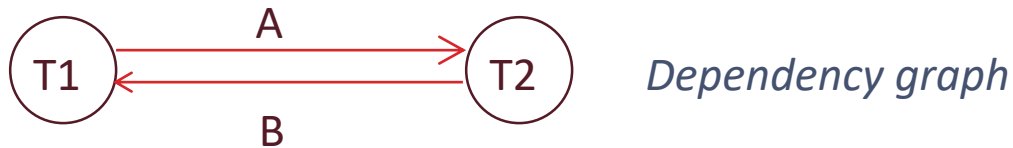
T1:	R(A), W(A),
T2:	R(A), W(A), R(B), W(B)



# Example, pt 4

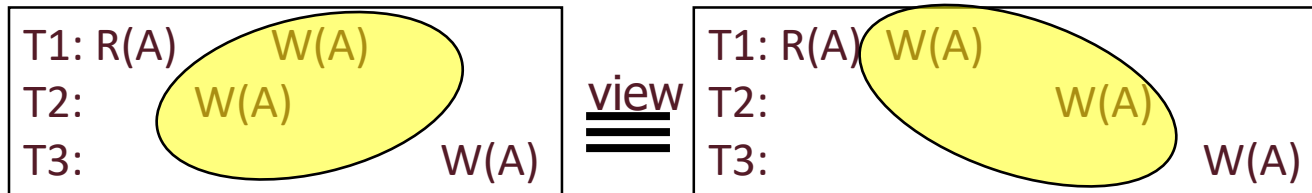
- **A schedule that is not conflict serializable**

T1:	R(A), W(A),	R(B)
T2:	R(A), W(A), R(B), W(B)	



# View Serializability

- **Alternative notion of serializability: fewer false negatives**
- **Schedules S1 and S2 are view equivalent if:**
  - *Same initial reads:*
    - If  $T_i$  reads initial value of A in S1, then  $T_i$  also reads initial value of A in S2
  - *Same dependent reads:*
    - If  $T_i$  reads value of A written by  $T_j$  in S1, then  $T_i$  also reads value of A written by  $T_j$  in S2
  - *Same winning writes:*
    - If  $T_i$  writes final value of A in S1, then  $T_i$  also writes final value of A in S2
- Basically, allows all conflict serializable schedules + “blind writes”





# Notes on Serializability Definitions

- **View Serializability allows (a few) more schedules than conflict serializability**
  - But V.S. is difficult to enforce efficiently.
- **Neither definition allows all schedules that are actually serializable.**
  - Because they don't understand the meanings of the operations or the data
- **Conflict Serializability is what gets used, because it can be enforced efficiently**
  - To allow more concurrency, some special cases do get handled separately.
  - (Search the web for "Escrow Transactions" for example)