Transactions & Concurrency Control 1

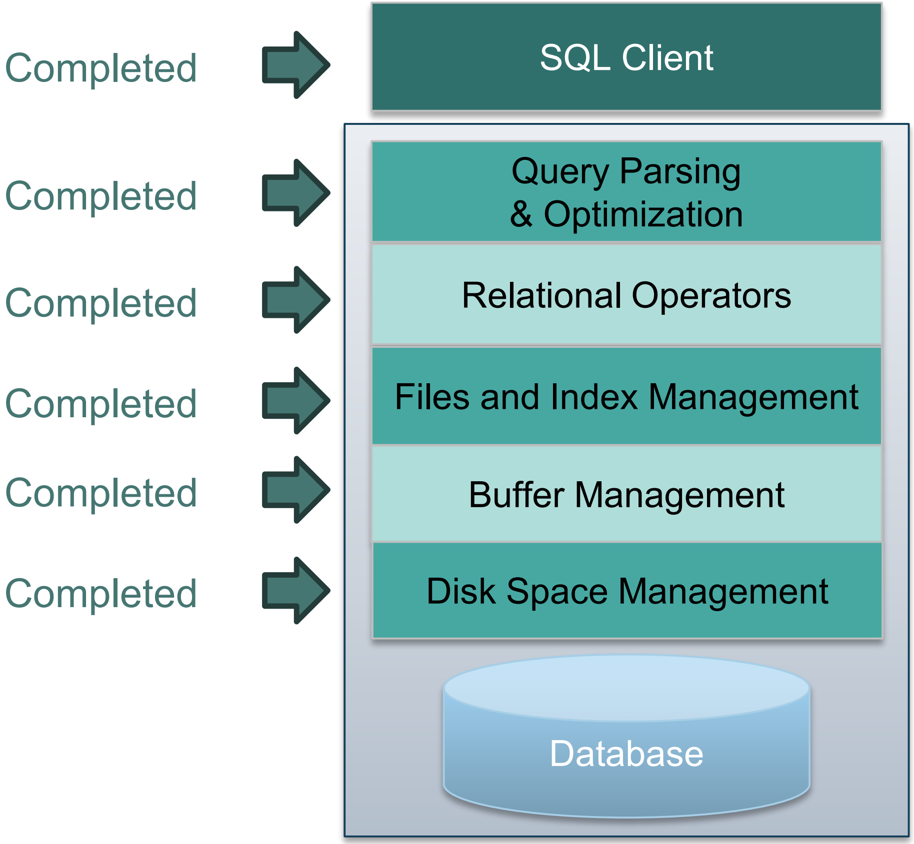
R&G 16/17

There are three side effects of acid. Enhanced long term memory, decreased short term memory, and I forget the third.

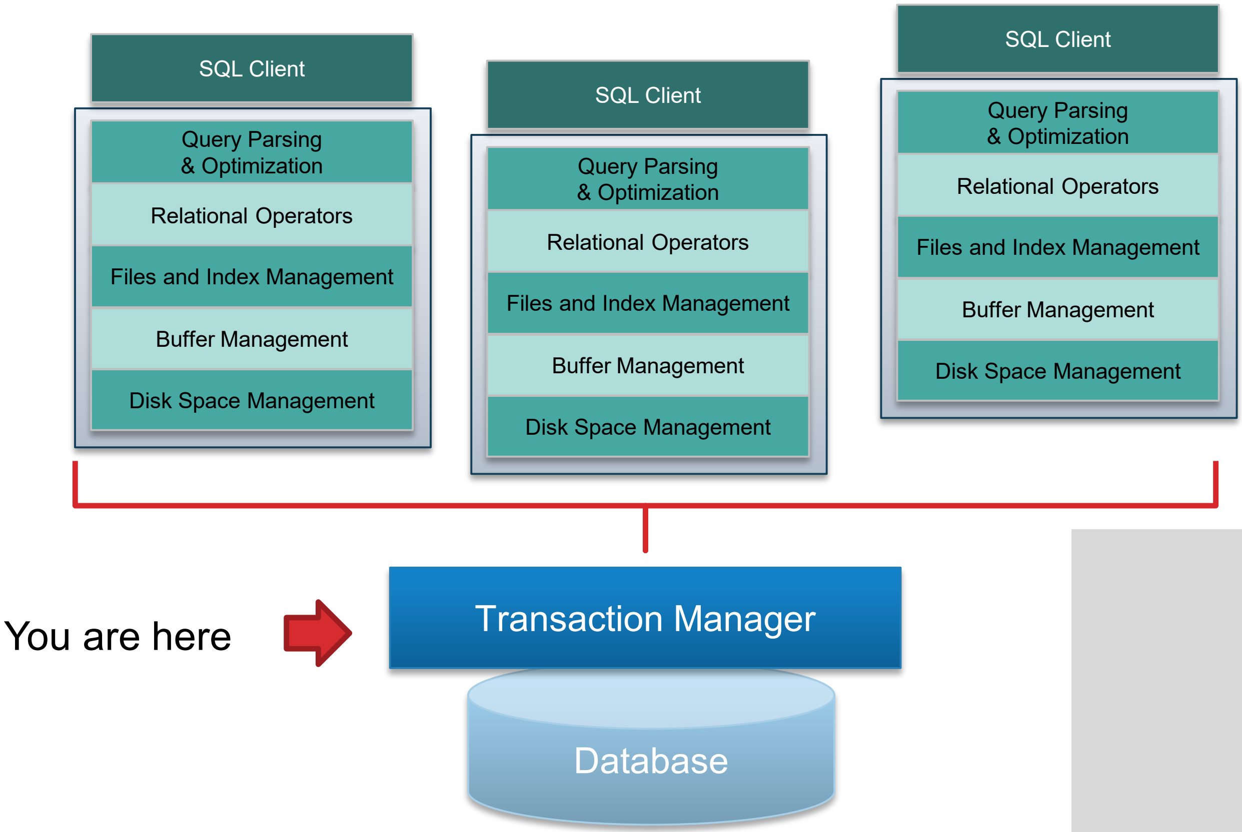
- Timothy Leary

Berkeley
cs186

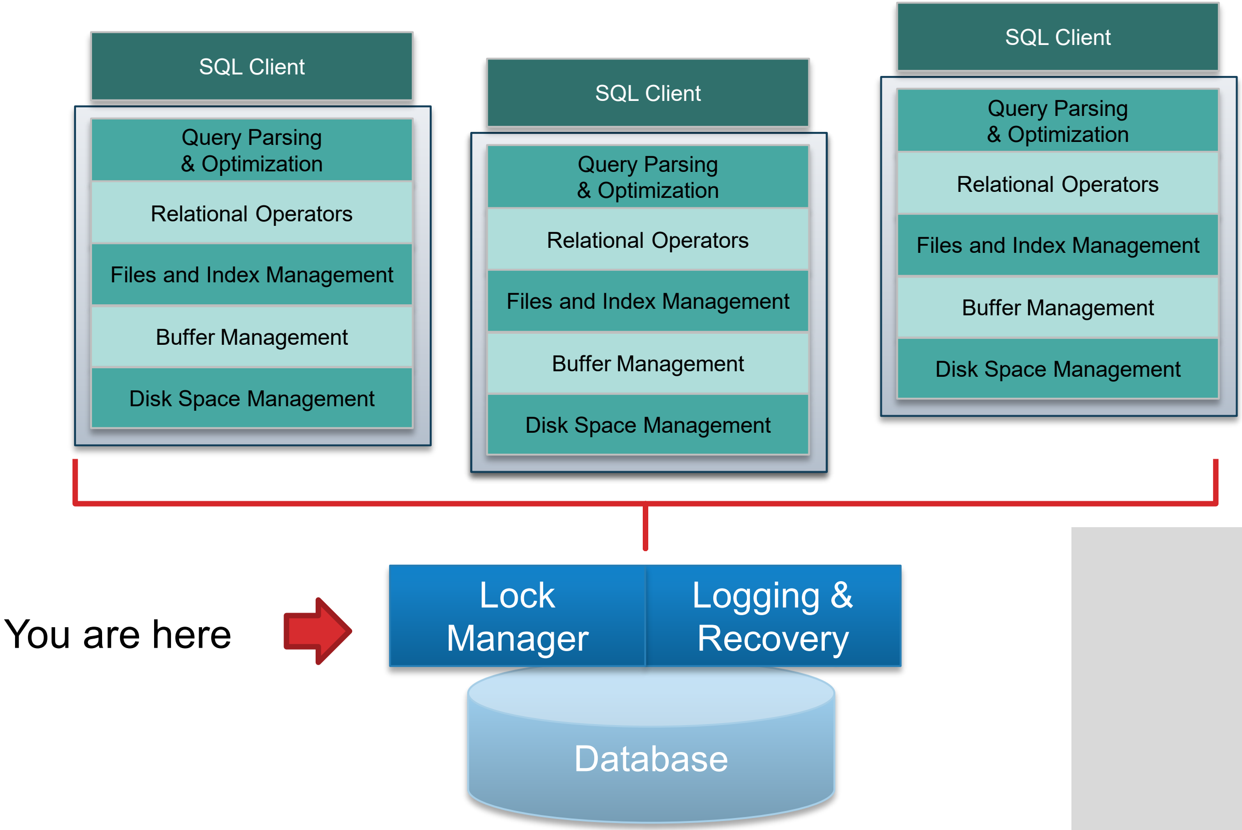
Architecture of a DBMS



Architecture of a DBMS, Part 2

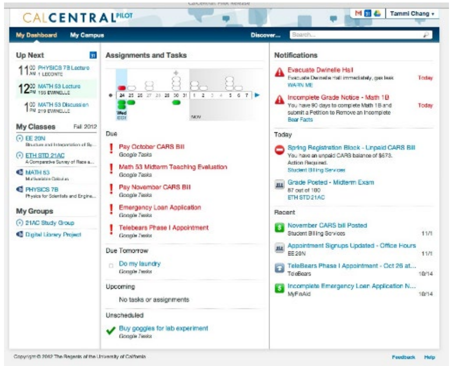


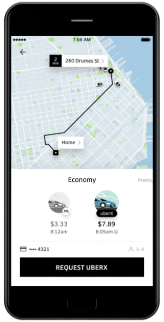
Architecture of a DBMS, Part 3

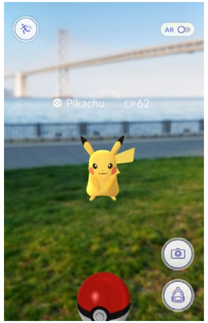


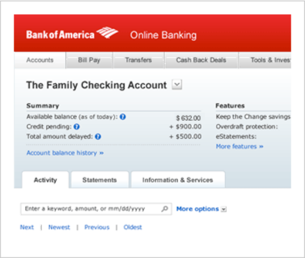
Applications on DBMS

- 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





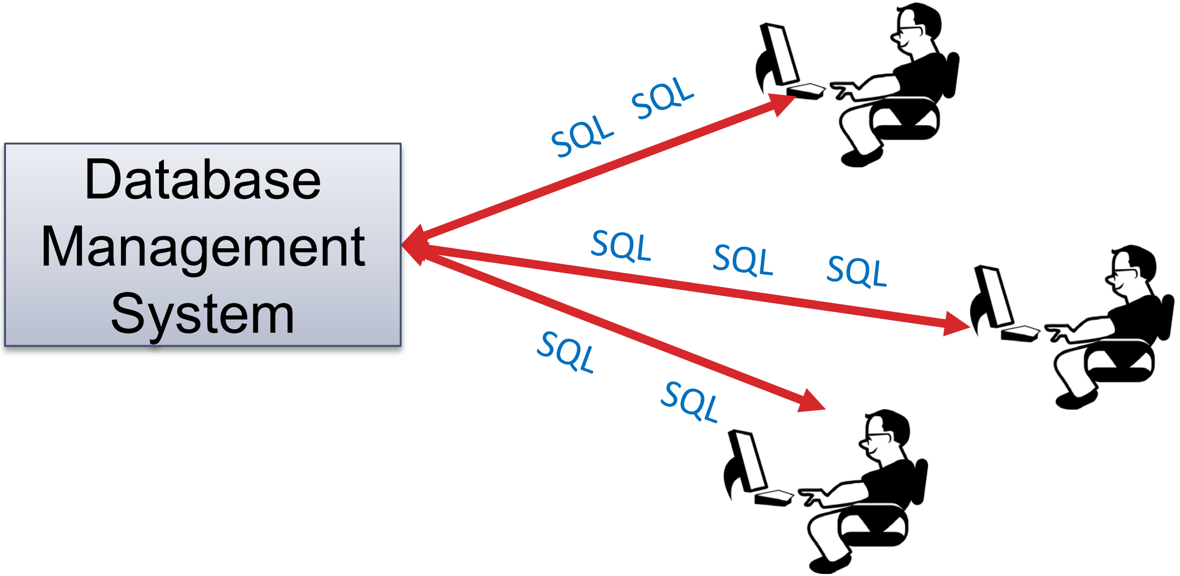




KAYAK

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”

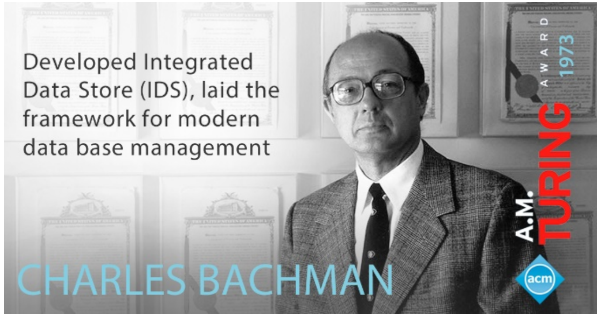
What could go wrong? **Dirty Reads**

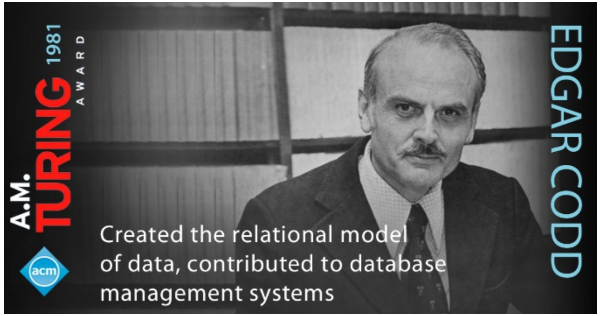
TRANSACTIONS

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

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

Not seen by the DBMS transaction manager!

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.

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

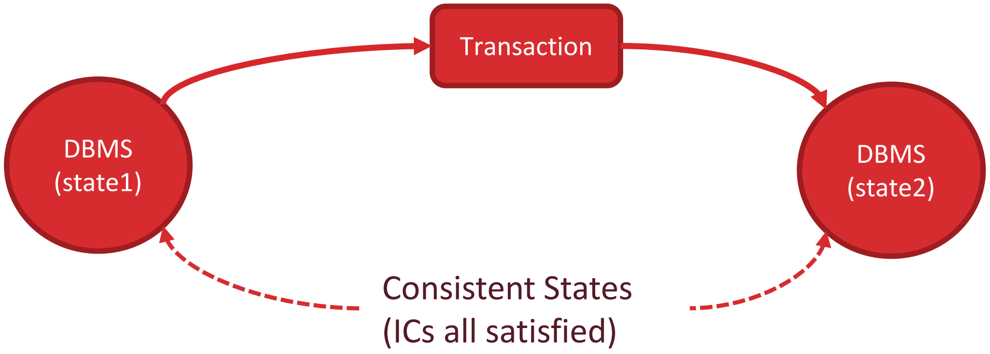
Atomicity and Durability, cont.

- 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 Consistency

- **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.

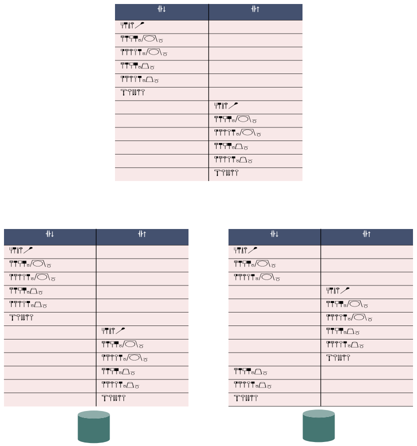
Actions: Begin, Read, Write, Commit and Abort.

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

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

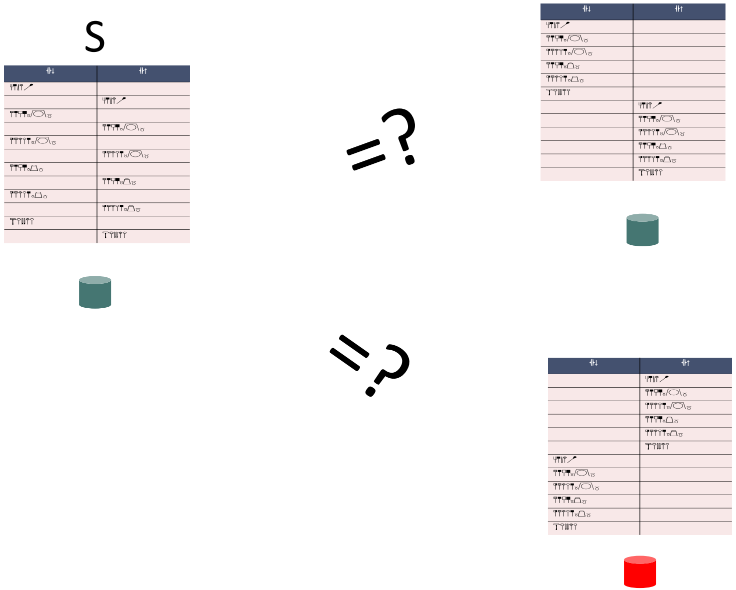
Serial Equivalence

- 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



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**!

Thanks to @jmpatel

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***

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

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***

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

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

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***

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

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

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***

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

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

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***

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

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

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

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

Conflict Serializability (Continued)

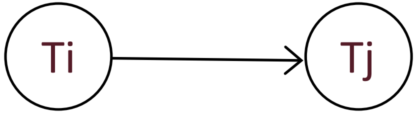
- **Here's another example:**

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

- **Conflict Serializable or not?**

NOT!

Conflict Dependency Graph



- **Dependency Graph:**
  - One node per Xact
  - Edge from Ti to Tj if:
    - An operation Oi of Ti conflicts with an operation Oj of Tj and
    - Oi appears earlier in the schedule than Oj

- **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)

Dependency graph

Example, pt 3

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

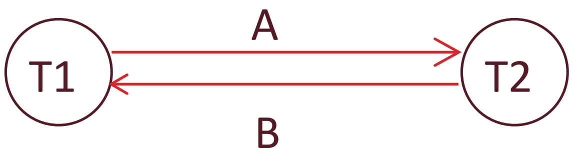
T1:	R(A), W(A),	
T2:		R(A), W(A), R(B), W(B)

Dependency graph

Example, pt 4

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

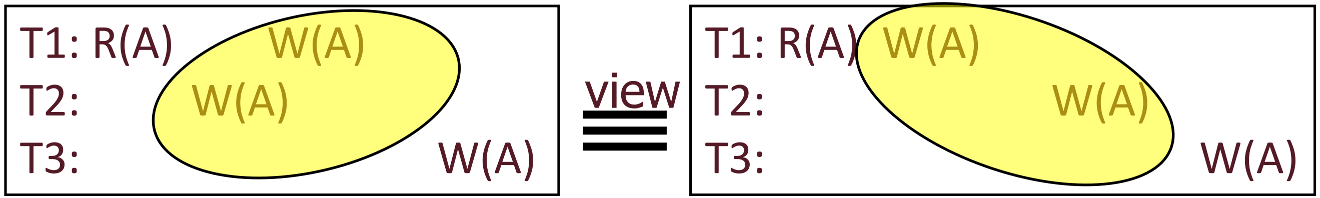
T1:	R(A), W(A),		R(B)
T2:		R(A), W(A), R(B), W(B)	



Dependency graph

View Serializability

- **Alternative notion of serializability: fewer false negatives**
- **Schedules S1 and S2 are view equivalent if:**
  - *Same initial reads*:
    - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - *Same dependent reads*:
    - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - *Same winning writes*:
    - If Ti writes final value of A in S1, then Ti 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)