CSE 541: Database Systems I

Transactions & Concurrency Control

Slides Credit: Andy Pavlo, Tianzheng Wang

Transactions

Definition: A transactions is a user program that runs in a DBMS to access (read/write) database records

- A transaction may
  - Access (read/write) one or multiple database records
  - Conduct computation with the records

```
BEGIN TRANSACTION
UPDATE Cities
SET Population = 1000000
WHERE City = Vancouver
COMMIT
```

- Desirable to run multiple transactions concurrently
  - Utilize the abundant parallelism provided by multicore CPUs
  - Keep CPU busy while I/O is in-progress

Concurrency vs. Parallelism?

Concurrency in DBMSs

- Multiple users can submit transactions to the DBMS
  - The DBMS will interleave read and write actions of different transactions
- A transaction must leave the database in a consistent state
  - DBMS enforces integrity constraints specified in table schema
  - DBMS doesn’t know data semantics

```
CREATE TABLE Cities (
    Id INTEGER,
    City CHAR(50),
    Population INTEGER,
    PRIMARY KEY(Id),
    CHECK (Population > 0)
)
```

Integrity constraint

ACID Properties

Ensured despite concurrent accesses and system failures

- **A**tomicity
  - Actions in a transaction are either all applied, or not at all
  - Commit – apply all read/write actions
  - Abort – rollback all changes so far, as if nothing happened
- **C**onsistency
  - Must leave the database in a consistent state
  - No data corruption, anomalies, etc. (more on this later)
- **I**solation
  - As if the transaction were the only one in the system
- **D**urability
  - Successful changes must be correctly persisted in storage

Example

Initial account balances:

Alice	Bob
100	100

Consider two transactions submitted **at the same time**

T1

```
BEGIN
Alice = Alice + 100
Bob = Bob – 100
COMMIT
```

Transfer 100 from Bob to Alice

T2

```
BEGIN
Alice = Alice * 1.05
Bob = Bob * 1.05
COMMIT
```

Credit a 5% interest on both accounts

- Actions are interleaved for high performance
- Many ways to interleave (schedules), but not all are valid

Transaction Schedules

Schedule: a list of actions from a set of transactions

- Actions: read, write, abort, commit
- Example:
  - T1: Read A, Write B, Commit ➔ denoted R1(A), W1(B), C1
  - T2: Write C, Read B, Commit ➔ denoted R2(C), W2(B), C2

  ➔A possible schedule: R1(A), R2(C), W1(B), C1, W2(B), C2
- Two actions of a transaction T that appear in the schedule, must be in the same order as they are in T
  - Consider the previous example
  - R1(A), R2(C), W1 (B), C1, W2(B), C2 ✓
  - W1(B), R2(C), R1 (A), C1, W2(B), C2 ✗

Transaction Schedules

Serial schedule: Schedule that does not interleave actions of different transactions

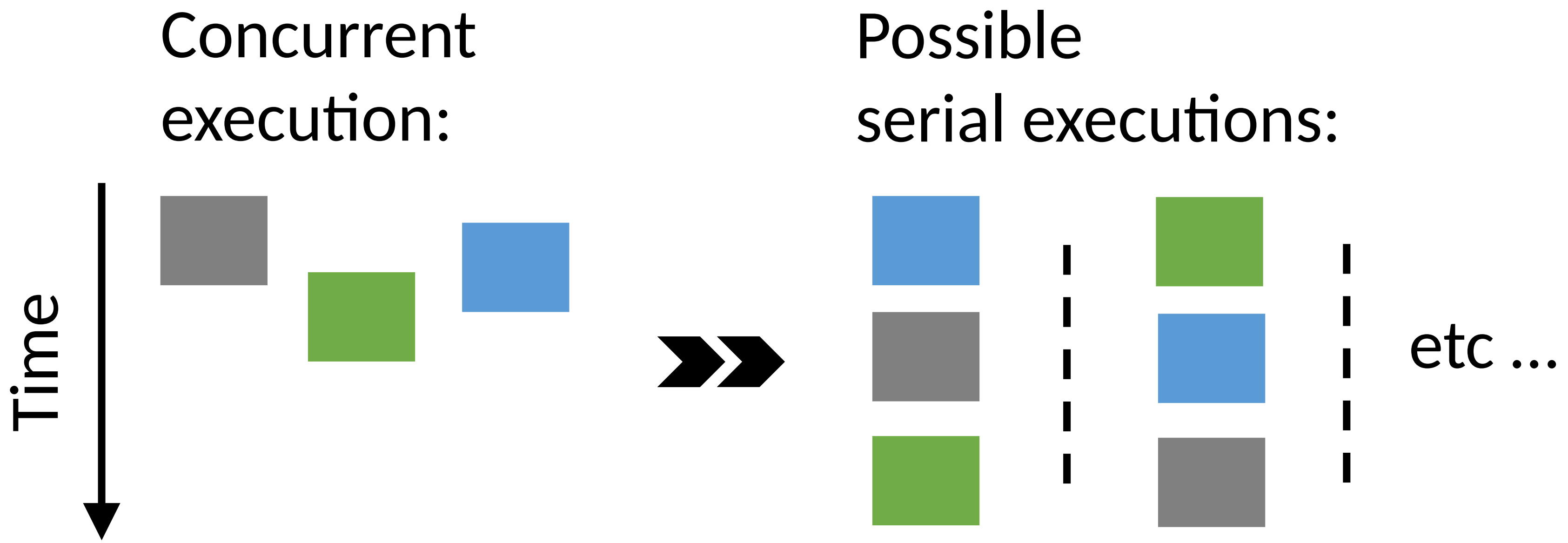
- Transactions executed one after another:
  - R1(A), W1(B), C1, R2(C), W2(B), C2 → T1, T2
  - R2(C), W2(B), C2, R1(A), W1(B), C1 → T2, T1

Equivalent schedules: two schedules are equivalent if their final states are the same

Serializability

Serializable schedule: A schedule that is equivalent to *some* serial execution's schedule (**doesn't matter which one**)

➔ A DBMS provides serializability if it produces serializable schedules



Example

Initial account balances:

Alice	Bob
100	100

Consider two transactions submitted **at the same time**

T1

```
BEGIN
Alice = Alice + 100
Bob = Bob – 100
COMMIT
```

Transfer 100 from Bob to Alice

T2

```
BEGIN
Alice = Alice * 1.05
Bob = Bob * 1.05
COMMIT
```

Credit a 5% interest on both accounts

Serializable: final effect must be equivalent to that of executing [T1 then T2] or [T2 then T1]

Schedule 1

Initial account balances:

Alice	Bob
100	100

Time	T1	T2
↓	`Alice = Alice + 100`	
		`Alice = Alice * 1.05`
	`Bob = Bob – 100`	
		`Bob = Bob * 1.05`

Final results:

Alice	Bob
210	0

Correct: as if we executed T1, then T2

Schedule 2

Initial account balances:

Alice	Bob
100	100

Time

T1	T2
	`Alice = Alice * 1.05`
`Alice = Alice + 100`	
	`Bob = Bob * 1.05`
`Bob = Bob – 100`	

Final results:

Alice	Bob
205	5

Correct: as if we executed T2, then T1

Schedule 3

Initial account balances:

Alice	Bob
100	100

T1

T2

Time

```
Alice = Alice + 100
```

```
Alice = Alice * 1.05
Bob = Bob * 1.05
```

```
Bob = Bob – 100
```

Final results:

Alice	Bob
210	5

Non-serializable schedule: no serial execution produces this result. A serializable concurrency control protocol is needed to prevent this.

Note on Serializability

- Guaranteeing serializability will ensure correct results
- But, **not guaranteeing serializability != wrong results**
  - i.e., lower isolation levels may be "good enough" to give correct results
  - Depends on the workload. Possible already no anomalies arise under a weaker isolation level, e.g., read committed.
  - Guaranteeing serializability incurs overhead, so it might be advantageous to use lower isolation levels
- Example: the classic TPC-C benchmark exhibits no anomalies (i.e., wrong results) under snapshot isolation, which is not serializable
  - More on this later

Conflict Serializability

Conflicts: In a schedule, two actions conflict if they operation on the same data record and at least one of them is a write

- Write-write conflict: two transactions both write R
- Read-write conflict: one reads R, the other writes R

Non-conflicting operations: order change does not affect final outcome

- Operations from different transactions and accessing different records, or reading the same record
- Consider schedule: R**1**(A), R**2**(C), W**1**(B), C**1**, W**2**(B), C**2**
  - Switching R**1**(A) and R**2**(C) will not change the final outcome

Conflicting operations: changing order leads to different final outcome, e.g., two writes

Conflict Serializability

Conflict equivalent schedules: Two schedules S1 and S2 are conflict equivalent iff S1 can be transformed to S2 by swapping adjacent non-conflicting operations

- Consider H1 and H2, transform H2 to H1
  - S1: R1(A) R2(C) W1(B) C1 W2(B) C2
  - S2: R1(A) W1(B) R2(C) W2(B) C1 C2

Conflict serializable: A schedule S that is conflict equivalent to a serial schedule

- S can be transformed by swapping non-conflicting operations to a serial schedule

Conflict Serializability

Necessary but not sufficient condition for serializability

- Every conflict serializable schedule is serializable, but not vice versa
- Example
  - H1: W1(A) W2(B) W1(B) W2(A) W3(B)
  - H2: W1(A) W1(B) W2(B) W2(A) W3(B)

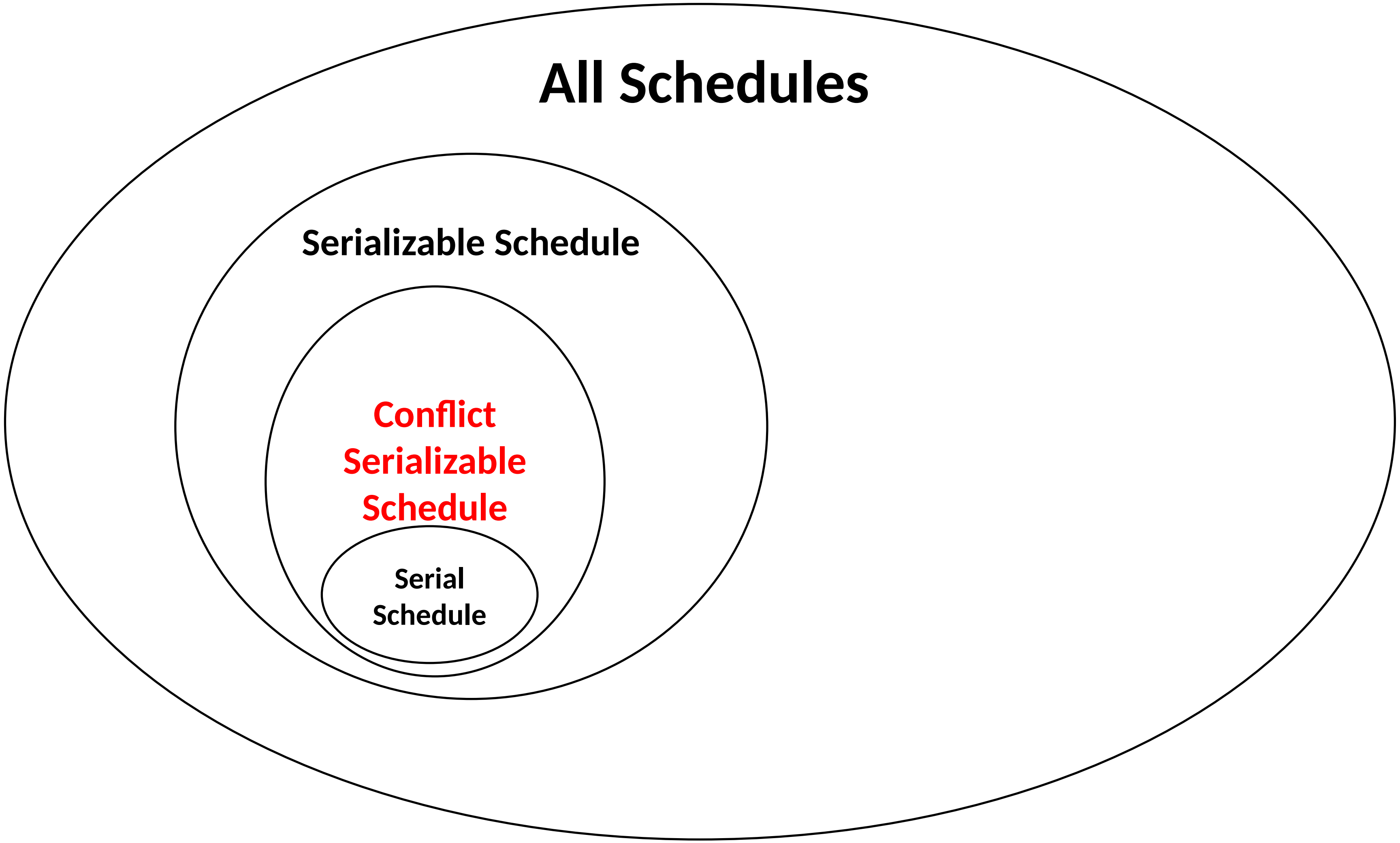
    T1 T2 T3

  Serial schedule

  - The final result of H1 and H2 are the same
    - Definition of serializability: final outcome same as some serial schedule's

    ➔ H1 is serializable!
  - But there are no way to transform H1 to H2 by swapping **adjacent, non-conflicting pairs**, so H1 is not conflict-serializable

Conflict Serializable Schedule

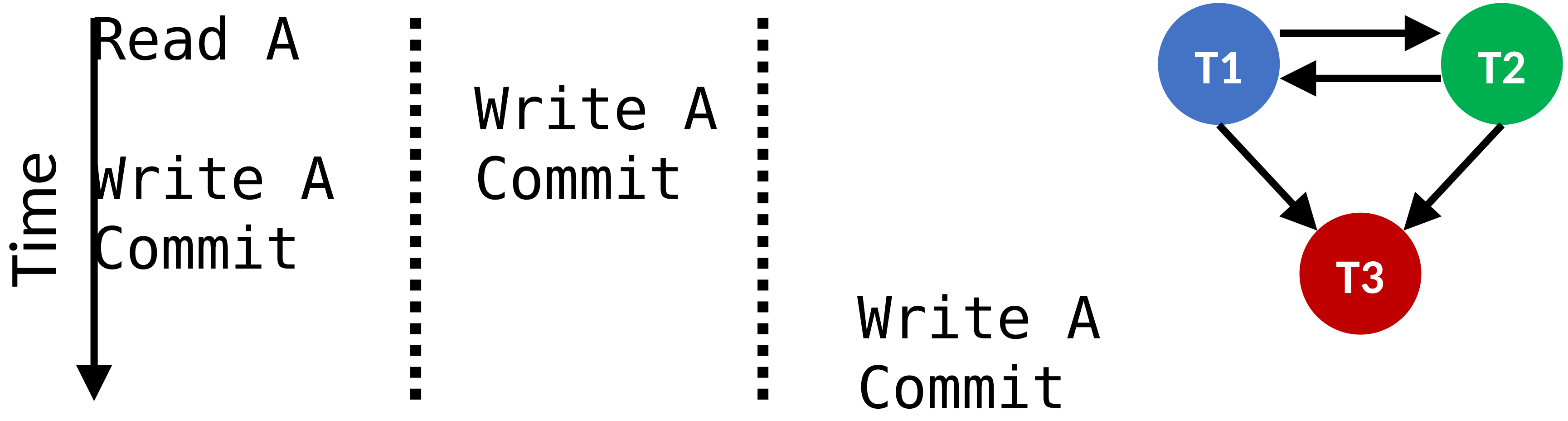


Dependency Graph

An effective way to capture potential conflicts between transactions in a schedule (aka “precedence graph”)

- Node: committed transaction
- Directed Edge: from T1 to T2 if T1 precedes and conflicts with one of the actions of T2

Schedule is conflict-serializable iff the dep. graph is <u>acyclic</u>



Anomalies

- Reading uncommitted data ("dirty reads")

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

- Unrepeatable Reads

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

- Overwriting committed data ("lost updates")

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

- Phantoms
  - The same range scan executed twice in the same transaction return different results

ANSI Isolation Levels

- Serializability gives the highest correctness guarantee
- There are also lower-level guarantees (may corrupt data or generate wrong results, depending on the workload)
- **Isolation levels defined by ANSI**
  - Read Uncommitted
  - Read Committed: no dirty reads
  - Repeatable Read: Read Committed + no non-repeatable reads
  - Serializable: Repeatable Reads + no phantom
- ANSI definitions are lock-based and not comprehensive
  - E.g., Snapshot Isolation isn’t truly serializable, yet does not produce any of the anomalies in ANSI SQL-92