>>

Transaction Isolation

/\ >>

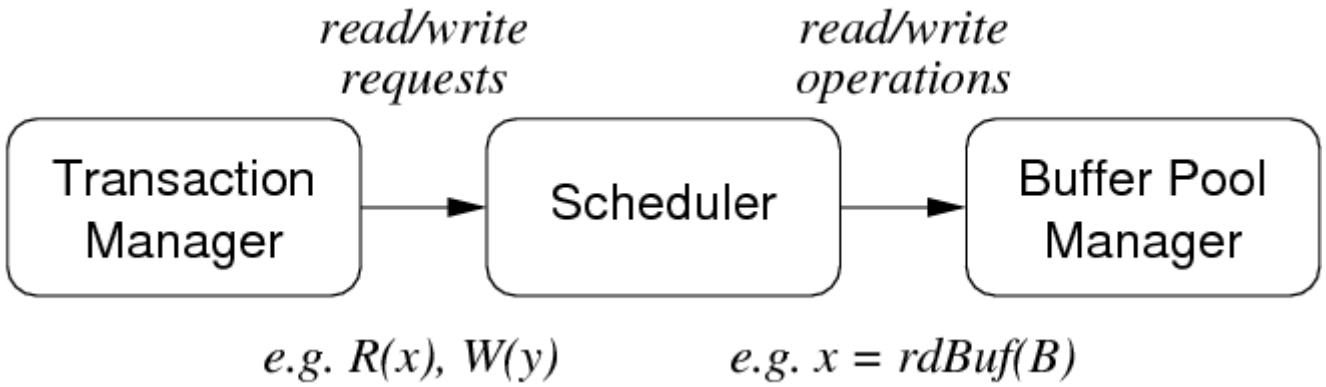
❖ Transaction Isolation

Simplest form of isolation: serial execution ($T_1; T_2; T_3; ...$)

Problem: serial execution yields poor throughput.

Concurrency control schemes (CCSs) aim for "safe" concurrency

Abstract view of DBMS concurrency mechanisms:



<< ^ >>

❖ Serializability

Consider two schedules S_1 and S_2 produced by

- executing the same set of transactions $T_1..T_n$ concurrently
- but with a non-serial interleaving of *R/W* operations

S_1 and S_2 are equivalent if $StateAfter(S_1) = StateAfter(S_2)$

- i.e. final state yielded by S_1 is same as final state yielded by S_2

S is a serializable schedule (for a set of concurrent tx's $T_1..T_n$) if

- S is equivalent to some serial schedule S_s of $T_1..T_n$

Under these circumstances, consistency is guaranteed
(assuming no aborted transactions and no system failures)

❖ Serializability (cont)

Two formulations of serializability:

- conflict serializibility
  - i.e. conflicting R/W operations occur in the "right order"
  - check via precedence graph; look for absence of cycles
- view serializibility
  - i.e. read operations *see* the correct version of data
  - checked via VS conditions on likely equivalent schedules

View serializability is strictly weaker than conflict serializability.

<< ^ >>

❖ Checking Serializability

Conflict serializablility checking:

```
make a graph with just nodes, one for each Ti
for each pair of operations across transactions {
    if (Ti and Tj have conflicting ops on variable X) {
        put a directed edge between Ti and Tj
            where the direction goes from
            first tx to access X to second tx to access X
        if this new edge forms a cycle in the graph
            return "Not conflict serializable"
    }
}
return "Conflict serializable"
```

<< ^ >>

❖ Checking Serializability (cont)

View serializability checking:

```
// T_C,i denotes transaction i in concurrent schedule
for each serial schedule S {
    // T_S,i denotes transaction i in serial schedule
    for each shared variable X {
        if T_C,i reads same version of X as T_S,i
            (either initial value or value written by T_j)
            continue
        else
            give up on this serial schedule
        if T_C,i and T_S,i write the final version of X
            continue
        else
            give up on this serial schedule
     }
     return "View serializable"
}
return "Not view serializable"
```

<< ^ >>

❖ Transaction Isolation Levels

SQL programmers' concurrency control mechanism ...

```
set transaction
    read only  -- so weaker isolation may be ok
    read write -- suggests stronger isolation needed
isolation level
    -- weakest isolation, maximum concurrency
    read uncommitted
    read committed
    repeatable read
    serializable
    -- strongest isolation, minimum concurrency
```

Applies to current tx only; affects how scheduler treats this tx.

<< ^ >>

❖ Transaction Isolation Levels (cont)

Implication of transaction isolation levels:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

<< ^ >>

❖ Transaction Isolation Levels (cont)

For transaction isolation, PostgreSQL

- provides syntax for all four levels
- treats read uncommitted as read committed
- repeatable read behaves *like* serializable
- default level is read committed

Note: cannot implement read uncommitted because of MVCC

For more details, see PostgreSQL Documentation section 13.2

- extensive discussion of semantics of **`UPDATE`**, **`INSERT`**, **`DELETE`**

<< ^ >>

❖ Transaction Isolation Levels (cont)

A PostgreSQL tx consists of a sequence of SQL statements:

```
BEGIN S1; S2; ... Sn; COMMIT;
```

Isolation levels affect view of DB provided to each S_i:

- in *read committed* ...
  - each S_i sees snapshot of DB at start of S_i
- in *repeatable read* and *serializable* ...
  - each S_i sees snapshot of DB at start of tx
  - serializable checks for extra conditions

Transactions fail if the system detects violation of isolation level.

<< ^ >>

❖ Transaction Isolation Levels (cont)

Example of repeatable read vs serializable

- table R(class,value) containing (1,10) (1,20) (2,100) (2,200)
- T1: X = sum(value) where class=1; insert R(2,X); commit
- T2: X = sum(value) where class=2; insert R(1,X); commit
- with repeatable read, both transactions commit, giving
  - updated table: (1,10) (1,20) (2,100) (2,200) (1,300) (2,30)
- with serial transactions, only one transaction commits
  - T1;T2 gives (1,10) (1,20) (2,100) (2,200) (2,30) (1,330)
  - T2;T1 gives (1,10) (1,20) (2,100) (2,200) (1,300) (2,330)
- PG recognises that committing both gives serialization violation

<< ^

❖ Concurrency Control

Isolation requires some method to control concurrency

Possible approaches to implementing concurrency control:

- Lock-based
  - Synchronise tx execution via locks on relevant part of DB.
- Version-based (multi-version concurrency control)
  - Allow multiple consistent versions of the data to exist.
    Each tx has access only to version existing at start of tx.
- Validation-based (optimistic concurrency control)
  - Execute all tx's; check for validity problems on commit.
- Timestamp-based
  - Organise tx execution via timestamps assigned to actions.

Produced: 14 Apr 2021