>>

Week 9 Online Sessions

- Week 09
- Transactions
- Exercise: Transactions
- Schedules
- Transaction Anomalies
- Exercise: Update Anomaly
- Exercise: How many Schedules?
- Schedule Properties
- Serializability Checking
- Exercise: Serializability Checking
- Serializability
- Concurrency Control
- Transaction Failure
- Recoverability
- Strictness
- Week 09
- Properties of Schedules
- Classes of Schedules
- Exercise: Recoverability/Serializability
- Lock-based Concurrency Control
- Two-Phase Locking
- Problems with Locking
- Deadlock
- Exercise: Deadlock Handling
- Locking in PostgreSQL
- Exercise: Using Locks
- Multi-version Concurrency Control
- Concurrency Control in PostgreSQL

❖ Week 09

Things to note for this week and next week and ...

- Assignment 1 marking queries all investigated
- Assignment 2 due date poll ... 97% say "extend"
- New Assignment 2 due date: Tuesday 19 April 9pm
- Quiz 5 due before Friday 22 April at 9pm
- MyExperience due before April 28 at midnight

In this session ...

- transactions, schedules, serializability, deadlock

<< ^ >>

❖ Transactions

A transaction is ...

- an application-level atomic operation
- implemented using multiple database operations
- moves the database from one consistent state to another

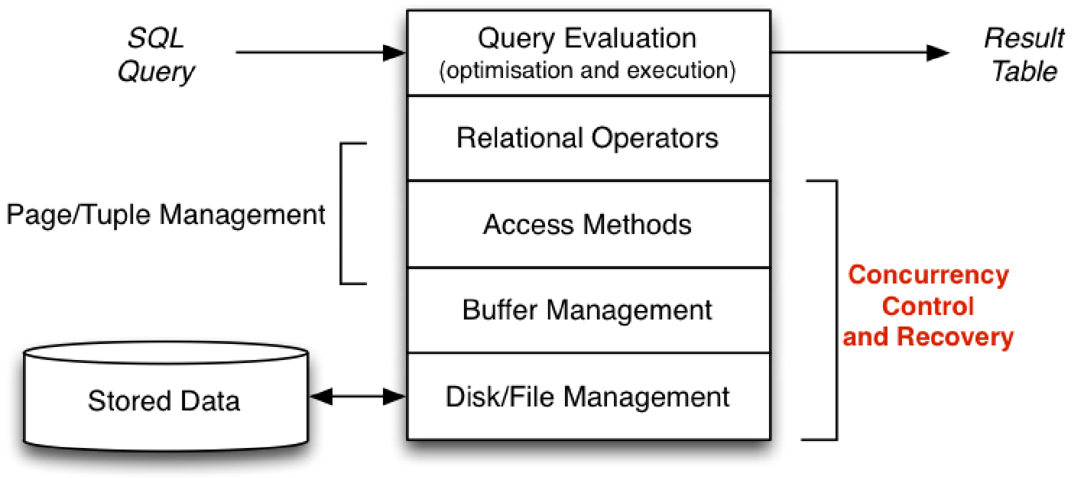
Beware: *during* transaction, database may not be consistent

Beware: multiple *concurrent* transactions may interfere

Note: each individual database operation is atomic

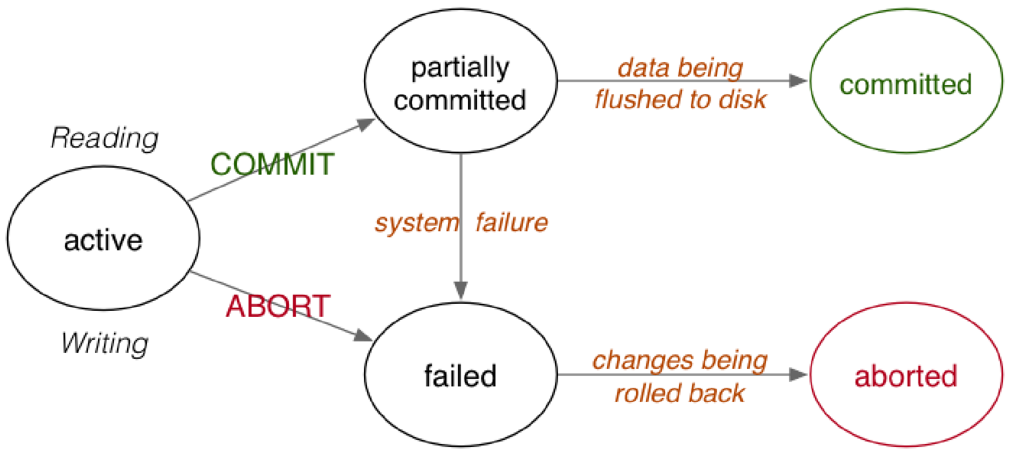
❖ Transactions (cont)

Where transaction processing fits in the DBMS:



❖ Transactions (cont)

Transaction states:



`COMMIT` ⇒ all changes preserved, **`ABORT`** ⇒ database unchanged

<< ∧ >>

❖ Transactions (cont)

Concurrent transactions are

- desirable, for improved performance (throughput)
- problematic, because of potential unwanted interactions

To ensure problem-free concurrent transactions:

- **A**tomic ... whole effect of tx, or nothing
- **C**onsistent ... individual tx's are "correct" (wrt application)
- **I**solated ... each tx behaves as if no concurrency
- **D**urable ... effects of committed tx's persist

❖ Transactions (cont)

Transaction processing:

- the study of techniques for realising ACID properties

Atomicity is handled by the commit and abort mechanisms

Consistency is handled by the programmer

Isolation is handled by concurrency control mechanisms (e.g. locking)

Durability is handled by implementing stable storage

❖ Exercise: Transactions

Consider the following transactions

- buying a ticket to a concert
- booking a seat on a flight

For each scenario ...

- suggest what a database schema might contain
- indicate what are the consistent before/after states
- suggest what DB operations would be involved
- show where problems might occur if transaction fails part-way
- show how two concurrent transactions might interfere

❖ Schedules

A schedule gives the sequence of operations from ≥ *1* tx

Serial schedule for a set of tx's $T_1 .. T_n$

- all operations of T_i complete before T_{i+1} begins

E.g. $R_{T_1}(A)$ $W_{T_1}(A)$ $R_{T_2}(B)$ $R_{T_2}(A)$ $W_{T_3}(C)$ $W_{T_3}(B)$

Concurrent schedule for a set of tx's $T_1 .. T_n$

- operations from individual T_i's are interleaved

E.g. $R_{T_1}(A)$ $R_{T_2}(B)$ $W_{T_1}(A)$ $W_{T_3}(C)$ $W_{T_3}(B)$ $R_{T_2}(A)$

<< ^ >>

❖ Schedules (cont)

Serial schedules guarantee database consistency

- each T_i commits before T_{i+1} starts
- prior to T_i database is consistent
- after T_i database is consistent (assuming T_i is correct)
- before T_{i+1} database is consistent ...

Concurrent schedules interleave tx operations arbitrarily

- and may produce a database that is not consistent
- after all of the transactions have committed successfully

❖ Transaction Anomalies

What problems can occur with (uncontrolled) concurrent tx's?

The set of phenomena can be characterised broadly under:

- dirty read:
  reading data item written by a concurrent uncommitted tx
- nonrepeateable read:
  re-reading data item, since changed by another concurrent tx
- phantom read:
  re-scanning result set, finding it changed by another tx

<< ^ >>

❖ Exercise: Update Anomaly

Consider the following transaction (expressed in pseudo-code):

```
-- Accounts(id,owner,balance,...)
transfer(src id, dest id, amount int)
{
    -- R(X)
    select balance from Accounts where id = src;
    if (balance >= amount) {
        -- R(X),W(X)
        update Accounts set balance = balance-amount
        where id = src;
        -- R(Y),W(Y)
        update Accounts set balance = balance+amount
        where id = dest;
}   }
```

If two transfers occur on this account simultaneously,
give a schedule that illustrates the "dirty read" phenomenon.

❖ Exercise: How many Schedules?

In the previous exercise, we looked at several schedules

For a given set of tx's $T_1 ... T_n$

- how many serial schedules are there?
- how many total schedules are there?

❖ Schedule Properties

If a concurrent schedule on a set of tx's *TT* ...

- produces the same effect as a serial schedule on *TT*
- then we say that the schedule is serializable

If a concurrent schedule on a set of tx's *TT* ...

- produces a consistent database state even if some tx's abort
- then we say that schedule is recoverable

❖ Serializability Checking

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.

❖ Exercise: Serializability Checking

Is the following schedule view/conflict serializable?

```
T1:        W(B)  W(A)
T2:  R(B)                    W(A)
T3:                    R(A)        W(A)
```

Is the following schedule view/conflict serializable?

```
T1:        W(B)  W(A)
T2:  R(B)              W(A)
T3:                          R(A)  W(A)
```

❖ Serializability

Knowing that a schedule for concurrent tx's

- produces the same result as some serial schedule

is useful to know, but not practical (i.e. after the event)

More important is scheduling tx operations, so that

- the overall effect is the same as some serial schedule

This forms the study of concurrency control

❖ Concurrency Control

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.

❖ Transaction Failure

So far, have implicitly assumed that all transactions commit.

Additional problems can arise when transactions abort.

Consider the following schedule where transaction T1 fails:

```
T1: R(X) W(X) A
T2:                R(X) W(X) C
```

Abort *will* rollback the changes to **X**, but ...

Consider three places where the rollback might occur:

```
T1: R(X) W(X) A [1]     [2]         [3]
T2:                 R(X)    W(X) C
```

<< ^ >>

❖ Recoverability

Consider the serializable schedule (ignoring **C** and **A**):

```
T1:        R(X)  W(Y)  C
T2:  W(X)                 A
```

(where the final value of **Y** is dependent on the **X** value)

Notes:

- the final value of X is valid (change from T_2 rolled back)
- T_1 reads/uses an X value that is eventually rolled-back
- even though T_2 is correctly aborted, it has produced an effect

Produces an invalid database state, even though serializable.

❖ Recoverability (cont)

Recoverable schedules avoid these kinds of problems.

For a schedule to be recoverable, we require additional constraints

- all tx's T_i that write values used by T_j must commit before T_j commits

and this property must hold for all transactions T_j

Note that recoverability does not prevent "dirty reads".

In order to make schedules recoverable in the presence of dirty reads and aborts, may need to abort multiple transactions.

❖ Strictness

A more constrained form of recoverability.

Strict schedules also eliminate the chance of *writing* dirty data.

A schedule is strict if

- no tx can read values written by another uncommitted tx (ACR)
- no tx can write a data item written by another uncommitted tx

Strict schedules simplify the task of rolling back after aborts.

❖ Week 09

Things to note for this week and next week and ...

- Assignment 1 marking queries all investigated
- Assignment 2 due date: Tuesday 19 April 9pm
  (I'm away over Easter; not able to deal with assignment queries)
- Quiz 5 due before Friday 22 April at 9pm
- MyExperience due before April 28 at midnight

In this session ...

- transactions, schedules, serializability, deadlock

<< ^ >>

❖ Properties of Schedules

Serializability

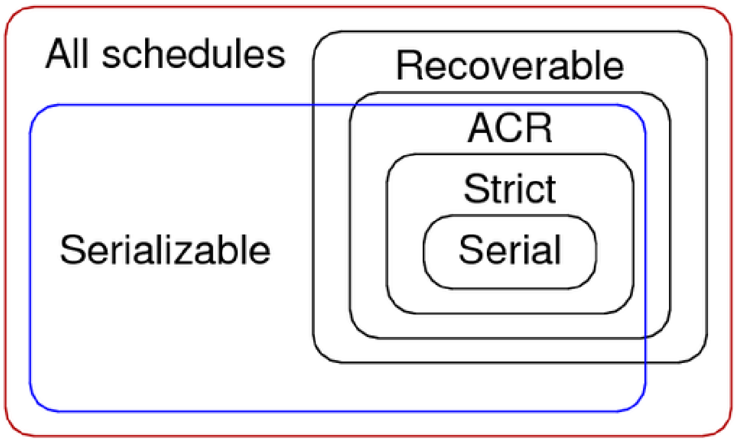
- schedule is *equivalent* to a serial schedule (SS)
- e.g. rearrange non-conflicting operations to yield SS

Recoverability (strictness)

- database consistency preserved even when tx's fail (abort)
- each tx commits only after each tx from which it has read commits

❖ Classes of Schedules

Relationship between various classes of schedules:



Schedules ought to be serializable and strict.

But more serializable/strict ⇒ less concurrency.

DBMSs allow users to trade off "safety" against performance.

❖ Exercise: Recoverability/Serializability

Recoverability and **S**erializability are orthogonal, i.e.

- a schedule can be R & S, !R & S, R &!S, !R & !S

Consider the two transactions:

```
T1:  W(A)  W(B)  C
T2:  W(A)  R(B)  C
```

Give examples of schedules on T1 and T2 that are

- recoverable and serializable
- not recoverable and serializable
- recoverable and not serializable

❖ Lock-based Concurrency Control

Requires read/write lock operations which act on database objects.

Synchronise access to shared DB objects via these rules:

- before reading *X*, get read (shared) lock on *X*
- before writing *X*, get write (exclusive) lock on *X*
- a tx attempting to get a read lock on *X* is blocked
  if another tx already has write lock on *X*
- a tx attempting to get a write lock on *X* is blocked
  if another tx has any kind of lock on *X*

These rules alone do not guarantee serializability.

Locks introduce additional mechanisms in DBMS (e.g. lock table)

<< ^ >>

❖ Lock-based Concurrency Control (cont)

Lock table entries contain:

- object being locked (DB, table, tuple, field)
- type of lock: read/shared, write/exclusive
- FIFO queue of tx's requesting this lock
- count of tx's currently holding lock (max 1 for write locks)

Lock and unlock operations *must* be atomic.

Lock upgrade:

- if a tx holds a read lock, and it is the only tx holding that lock
- then the lock can be converted into a write lock

❖ Lock-based Concurrency Control (cont)

Consider the following schedule, using locks:

```
T1: L_r(Y)    R(Y)                U(Y)         L_w(X) W(X) U(X)
T2:      L_r(X)   R(X) U(X)  L_w(Y)....W(Y) U(Y)
```

(where L_r = read-lock, L_w = write-lock, U = unlock)

Locks correctly ensure controlled access to **X** and **Y**.

Despite this, the schedule is not serializable

```
T1: R(Y)              W(X)
T2:       R(X)  W(Y)
```

<< ^ >>

❖ Two-Phase Locking

To guarantee serializability, we require an additional constraint:

- in every tx, all *lock* requests precede all *unlock* requests

Each transaction is then structured as:

- growing phase where locks are acquired
- action phase where "real work" is done
- shrinking phase where locks are released

Clearly, this reduces potential concurrency ...

❖ Problems with Locking

Appropriate locking can guarantee serializability.

However, it also introduces potential undesirable effects:

- Deadlock
    - No transactions can proceed; each waiting on lock held by another.
- Starvation
    - One transaction is permanently "frozen out" of access to data.
- Reduced performance
    - Locking introduces delays while waiting for locks to be released.

❖ Deadlock

Deadlock occurs when two tx's wait for a lock on an item held by the other.

Example:

```
T1: $L_W$(A) R(A)                 $L_W$(B) ......
T2:                $L_W$(B) R(B)         $L_W$(A) .....
```

How to deal with deadlock?

- prevent it happening in the first place
- let it happen, detect it, recover from it

<< ^ >>

❖ Deadlock (cont)

Handling deadlock involves forcing a transaction to "back off"

- select process to roll back
  - choose on basis of how far tx has progressed, # locks held, ...
- roll back the selected process
  - how far does this it need to be rolled back?
  - worst-case scenario: abort one transaction, retry later
- prevent starvation
  - need methods to ensure that same tx isn't always chosen

❖ Deadlock (cont)

Methods for managing deadlock

- timeout : set max time limit for each tx
- waits-for graph : records T_j waiting on lock held by T_k
  - *prevent* deadlock by checking for new cycle ⇒ abort T_i
  - *detect* deadlock by periodic check for cycles ⇒ abort T_i
- timestamps : use tx start times as basis for priority

❖ Exercise: Deadlock Handling

Consider the following schedule on four transactions:

```
T1: R(A)     W(C)                             W(D)
T2:     R(B)                      W(C)
T3:              R(D)      W(B)
T4:                   R(E)              W(A)
```

Assume that: each **R** acquires a shared lock; each **W** uses an exclusive lock; two-phase locking is used.

Show how the waits-for graph for the locks evolves.

Show how any deadlocks might be resolved via this graph.

❖ Locking in PostgreSQL

PostgreSQL provides two types of explicit locking

`LOCK TABLE` *TableName* **`IN`** *Mode* **`MODE`**

- mode can be **`SHARED`** or **`EXCLUSIVE`**

`SELECT` ... **`WHERE`** *Condition* **`FOR UPDATE`**

- applies exclusive locks to all matching rows

After exclusive lock, can make uninterrupted changes to table/rows

❖ Exercise: Using Locks

How could we solve this problem via locking?

```
create or replace function
    allocSeat(paxID int, fltID int, seat text)
    returns boolean
as $$
declare
    pid int;
begin
    select paxID into pid from SeatingAlloc
    where  flightID = fltID and seatNum = seat;
    if (pid is not null) then
        return false;  -- someone else already has seat
    else
        update SeatingAlloc set pax = paxID
        where  flightID = fltID and seatNum = seat;
        commit;
        return true;
    end if;
end;
$$ langauge plpgsql;
```

<< ^ >>

❖ Multi-version Concurrency Control

Multi-version concurrency control (MVCC) aims to

- retain benefits of locking, while getting more concurrency
- by providing multiple (consistent) versions of data items

Achieves this by

- readers access an "appropriate" version of each data item
- writers make new versions of the data items they modify

Main difference between MVCC and standard locking:

- read locks do not conflict with write locks ⇒
- reading never blocks writing, writing never blocks reading

❖ Multi-version Concurrency Control (cont)

WTS = timestamp of tx that wrote this data item

Chained tuple versions: $tup_{oldest} \rightarrow \ldots \rightarrow tup_{newest}$

When a reader T_i is accessing the database

- ignore any data item D written after T_i started
  - checked by: WTS(D) > TS(T_i)
- use only newest version V accessible to T_i
  - determined by: max(WTS(V)) < TS(T_i)

❖ Multi-version Concurrency Control (cont)

When a writer T_i attempts to change a data item

- find newest version V satisfying WTS(V) < TS(T_i)
- if no later versions exist, create new version of data item
- if there are later versions, then abort T_i

<< ^ >>

❖ Multi-version Concurrency Control (cont)

Advantage of MVCC

- locking needed for serializability considerably reduced

Disadvantages of MVCC

- visibility-check overhead (on every tuple read/write)
- storage overhead for extra versions of data items
- overhead in removing out-of-date versions of data items

Despite apparent disadvantages, MVCC is very effective.

❖ Multi-version Concurrency Control (cont)

Removing old versions:

- V_j and V_k are versions of same item
- $WTS(V_j)$ and $WTS(V_k)$ precede $TS(T_i)$ for all T_i
- remove version with smaller $WTS(V_x)$ value

When to make this check?

- every time a new version of a data item is added?
- periodically, with fast access to blocks of data

PostgreSQL uses the latter (vacuum).

❖ Concurrency Control in PostgreSQL

PostgreSQL uses two styles of concurrency control:

- multi-version concurrency control (MVCC)
  (used in implementing SQL DML statements (e.g. **`select`**))
- two-phase locking (2PL)
  (used in implementing SQL DDL statements (e.g. **`create table`**))

From the SQL (PLpgSQL) level:

- can let the lock/MVCC system handle concurrency
- can handle it explicitly via **`LOCK`** statements

❖ Concurrency Control in PostgreSQL (cont)

PostgreSQL provides read committed and serializable isolation levels.

Using the serializable isolation level, a **`select`**:

- sees only data committed before the transaction began
- never sees changes made by concurrent transactions

Using the serializable isolation level, an update fails:

- if it tries to modify an "active" data item
  (active = affected by some other tx, either committed or uncommitted)

The transaction containing the update must then rollback and re-start.

❖ Concurrency Control in PostgreSQL (cont)

Implementing MVCC in PostgreSQL requires:

- a log file to maintain current status of each T_i
- in every tuple:
  - **`xmin`** = ID of the tx that created the tuple
  - **`xmax`** = ID of the tx that replaced/deleted the tuple (if any)
  - **`xnew`** = link to newer versions of tuple (if any)
- for each transaction T_i:
  - a transaction ID (timestamp)
  - SnapshotData: list of active tx's when T_i started

❖ Concurrency Control in PostgreSQL (cont)

Rules for a tuple to be visible to T_i:

- the **`xmin`** (creation transaction) value must
  - be committed in the log file
  - have started before T_i's start time
  - not be active at T_i's start time
- the **`xmax`** (delete/replace transaction) value must
  - be blank or refer to an aborted tx, or
  - have started after T_i's start time, or
  - have been active at SnapshotData time

For details, see: **`backend/access/heap/heapam_visibility.c`**

<< ^

❖ Concurrency Control in PostgreSQL (cont)

Tx's always see a consistent version of the database.

But may not see the "current" version of the database.

E.g. T1 does select, then concurrent T2 deletes some of T1's selected tuples

This is OK unless tx's communicate outside the database system.

E.g. T1 counts tuples; T2 deletes then counts; then counts are compared

Use locks if application needs every tx to see same current version

- **`LOCK TABLE`** locks an entire table
- **`SELECT FOR UPDATE`** locks only the selected rows

Produced: 14 Apr 2022