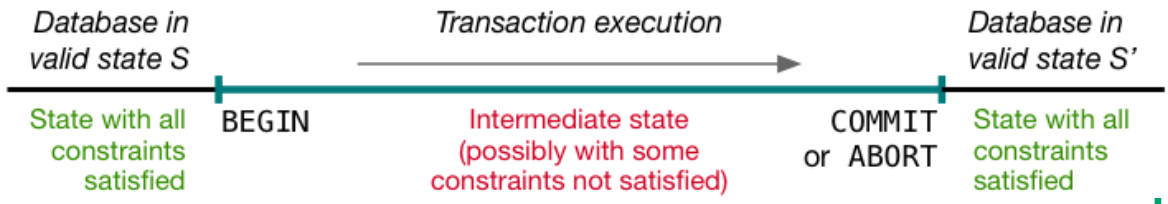
Transaction Processing

❖ Transaction Processing

A transaction (tx) is ...

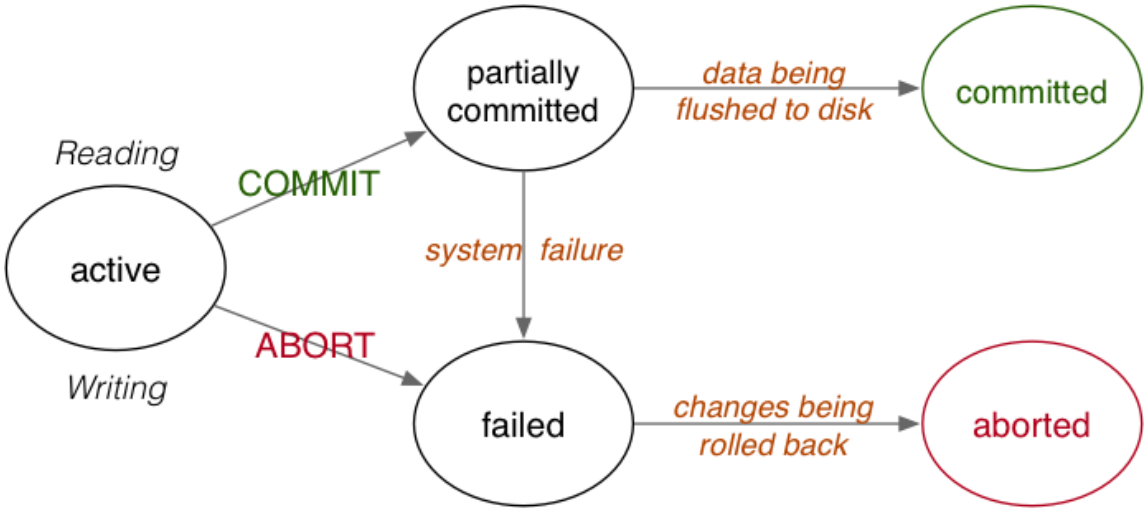
- a single application-level operation
- performed by a sequence of database operations

A transaction effects a state change on the DB



❖ Transaction Processing (cont)

Transaction states:



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

❖ Transaction Processing (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

❖ Transaction Processing (cont)

Transaction processing:

- the study of techniques for realising ACID properties

Consistency is the property:

- a tx is correct with respect to its own specification
- a tx performs a mapping that maintains all DB constraints

Ensuring this must be left to application programmers.

Our discussion focusses on: Atomicity, Durability, Isolation

❖ Transaction Processing (cont)

Atomicity is handled by the commit and abort mechanisms

- **commit** ends tx and ensures all changes are saved
- **abort** ends tx and *undoes* changes "already made"

Durability is handled by implementing stable storage, via

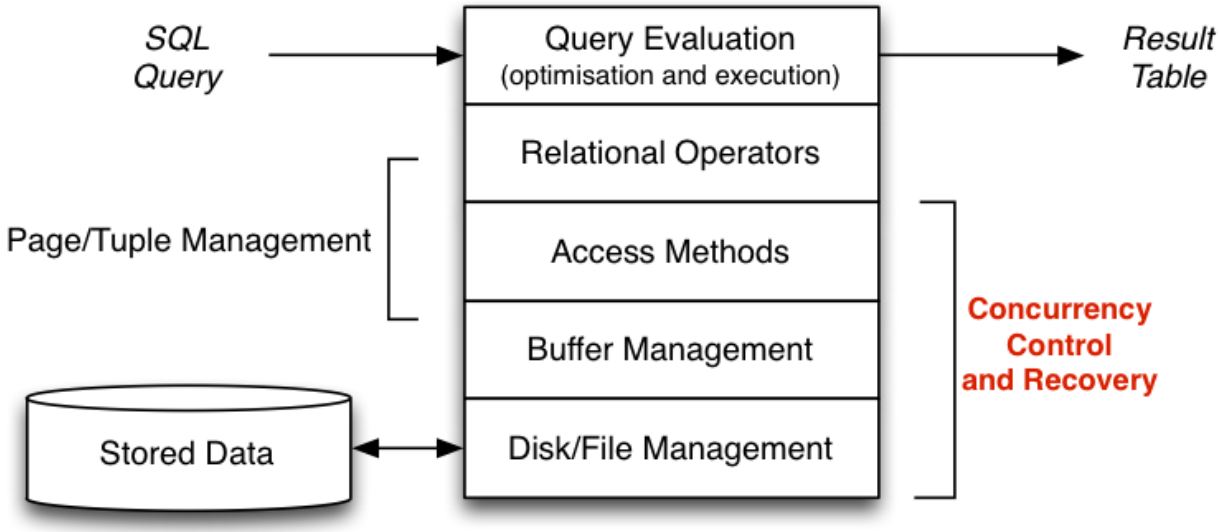
- redundancy, to deal with hardware failures
- logging/checkpoint mechanisms, to recover state

Isolation is handled by concurrency control mechanisms

- possibilities: lock-based, timestamp-based, check-based
- various levels of isolation are possible (e.g. serializable)

❖ Transaction Processing (cont)

Where transaction processing fits in the DBMS:



❖ Transaction Terminology

To describe transaction effects, we consider:

- **READ** - transfer data from "disk" to memory
- **WRITE** - transfer data from memory to "disk"
- **ABORT** - terminate transaction, unsuccessfully
- **COMMIT** - terminate transaction, successfully

Relationship between the above operations and SQL:

- **SELECT** produces **READ** operations on the database
- **UPDATE** and **DELETE** produce **READ** then **WRITE** operations
- **INSERT** produces **WRITE** operations

❖ Transaction Terminology (cont)

More on transactions and SQL

- **BEGIN** starts a transaction
  - the **begin** keyword in PLpgSQL is not the same thing
- **COMMIT** commits and ends the current transaction
  - some DBMSs e.g. PostgreSQL also provide **END** as a synonym
  - the **end** keyword in PLpgSQL is not the same thing
- **ROLLBACK** aborts the current transaction, undoing any changes
  - some DBMSs e.g. PostgreSQL also provide **ABORT** as a synonym

In PostgreSQL, tx's cannot be defined inside functions (e.g. PLpgSQL)

❖ Transaction Terminology (cont)

The **READ**, **WRITE**, **ABORT**, **COMMIT** operations:

- occur in the context of some transaction *T*
- involve manipulation of data items *X, Y, ...* (READ and WRITE)

The operations are typically denoted as:

$R_T(X)$	read item *X* in transaction *T*
$W_T(X)$	write item *X* in transaction *T*
A_T	abort transaction *T*
C_T	commit transaction *T*

❖ 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

Properties of Schedules

- Schedule Properties
- Serializable Schedules
- Transaction Failure
- Recoverability
- Cascading Aborts
- Strictness
- Classes of Schedules

❖ 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

A goal of isolation mechanisms (see later) is

- arrange execution of individual operations in tx's in *TT*
- to ensure that a serializable schedule is produced

Serializability is one property of a schedule, focusing on isolation

Other properties of schedules focus on recovering from failures

❖ Serializable Schedules

Producing a serializable schedule

- eliminates all update anomalies
- may reduce opportunity for concurrency
- may reduce overall throughput of system

If DB programmers know update anomalies are unlikely/tolerable

- serializable schedules may not be neccessary
- some DBMSs offer less strict isolation levels (e.g. repeatable read)
- allowing more opportunity for concurrency

❖ 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
```

❖ Transaction Failure (cont)

Abort / rollback scenarios:

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

Case [1] is ok

- all effects of T1 vanish; final effect is simply from T2

Case [2] is problematic

- some of T1's effects persist, even though T1 aborted

Case [3] is also problematic

- T2's effects are lost, even though T2 committed

❖ Recoverability

Consider the serializable schedule:

```
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.

❖ Cascading Aborts

Recall the earlier non-recoverable schedule:

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

To make it recoverable requires:

- delaying T_1's commit until T_2 commits
- if T_2 aborts, cannot allow T_1 to commit

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

Known as cascading aborts (or cascading rollback).

❖ Cascading Aborts (cont)

Example: T_3 aborts, causing T_2 to abort, causing T_1 to abort

```
T1:                          R(Y)  W(Z)        A
T2:          R(X)  W(Y)                   A
T3:  W(X)                              A
```

Even though T_1 has no direct connection with T_3 (i.e. no shared data).

This kind of problem ...

- can potentially affect very many concurrent transactions
- could have a significant impact on system throughput

❖ Cascading Aborts (cont)

Cascading aborts can be avoided if

- transactions can only read values written by committed transactions

  (alternative formulation: no tx can read data items written by an uncommitted tx)

Effectively: eliminate the possibility of reading dirty data

Downside: reduces opportunity for concurrency

GUW call these ACR (avoid cascading rollback) schedules.

All ACR schedules are also recoverable.

❖ Strictness

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.

❖ Strictness (cont)

Example: non-strict schedule

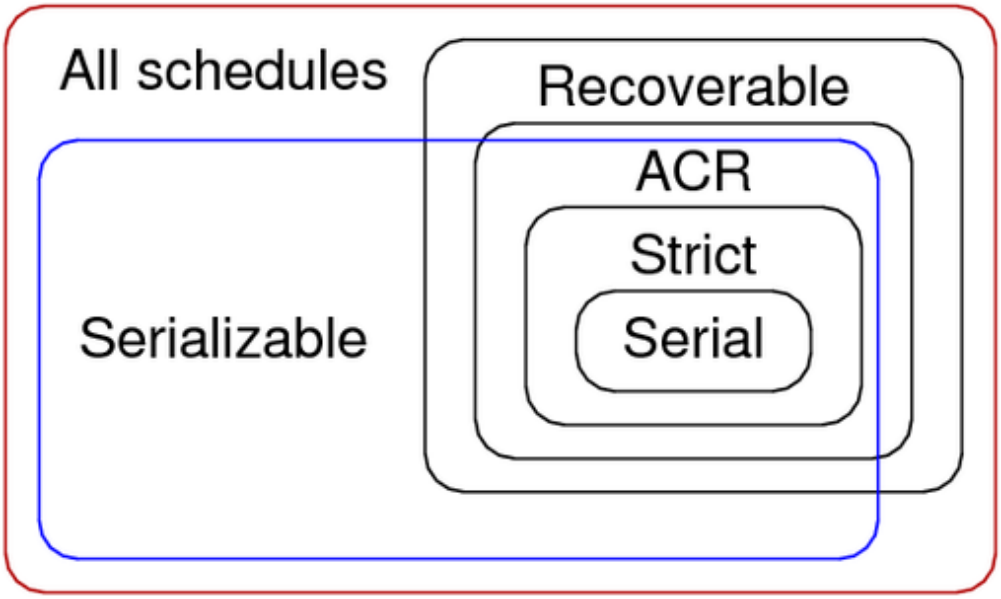
```
T1:  W(X)          A
T2:        W(X)      A
```

Problems with handling rollback after aborts:

- when T_1 aborts, don't rollback (need to retain value written by T_2)
- when T_2 aborts, need to rollback to pre-T_1 (not just pre-T_2)

❖ 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.

Transaction Isolation

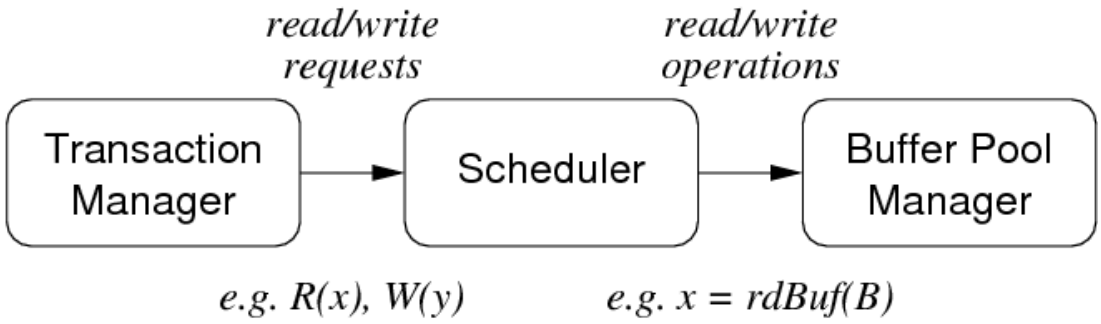
❖ Transaction Isolation

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.

Lock-based Concurrency Control

❖ Lock-based Concurrency Control

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

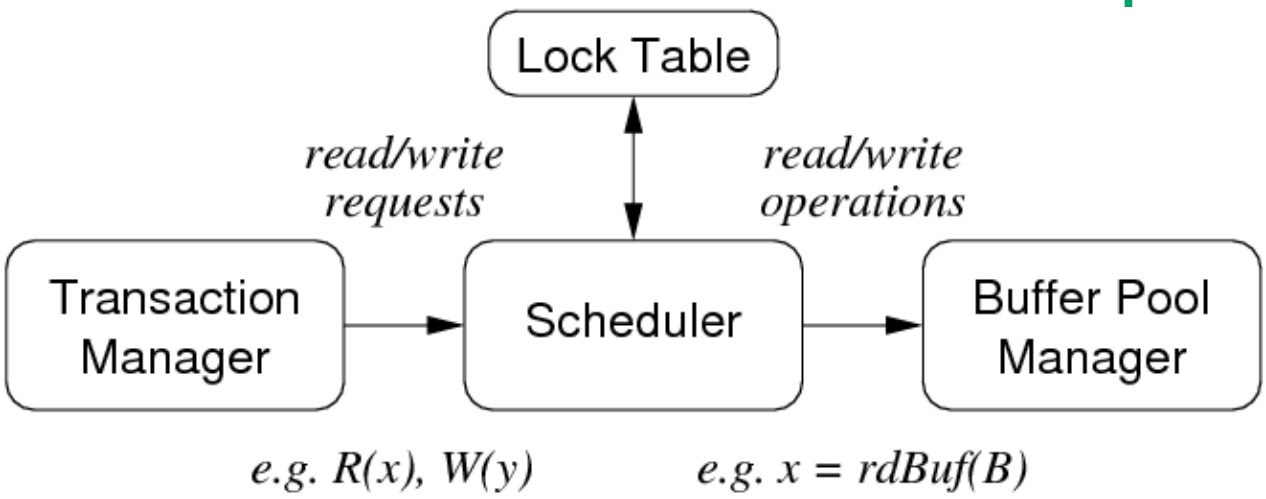
Synchronise access to shared DB objects via these rules:

- 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.

❖ Lock-based Concurrency Control (cont)

Locks introduce additional mechanisms in DBMS:



The Lock Manager manages the locks requested by the scheduler

❖ 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(a): Lr(Y)      R(Y)                continued
T2(a):      Lr(X)     R(X) U(X)       continued

T1(b):      U(Y)           Lw(X) W(X) U(X)
T2(b): Lw(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.
(Ex: prove this)

❖ 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, then retry
- 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
  - scenario: T_j tries to get lock held by T_k

    ...
  - wait-die: if $T_j < T_k$, then T_j waits, else T_j rolls back
  - wound-wait: if $T_j < T_k$, then T_k rolls back, else T_j waits

❖ Deadlock (cont)

Properties of deadlock handling methods:

- both wait-die and wound-wait are fair
- wait-die tends to
  - roll back tx's that have done little work
  - but rolls back tx's more often
- wound-wait tends to
  - roll back tx's that may have done significant work
  - but rolls back tx's less often
- timestamps easier to implement than waits-for graph
- waits-for minimises roll backs because of deadlock

Optimistic Concurrency Control

❖ Optimistic Concurrency Control

Locking is a pessimistic approach to concurrency control:

- limit concurrency to ensure that conflicts don't occur

Costs: lock management, deadlock handling, contention.

In scenarios where there are far more reads than writes ...

- don't lock (allow arbitrary interleaving of operations)
- check just before commit that no conflicts occurred
- if problems, roll back conflicting transactions

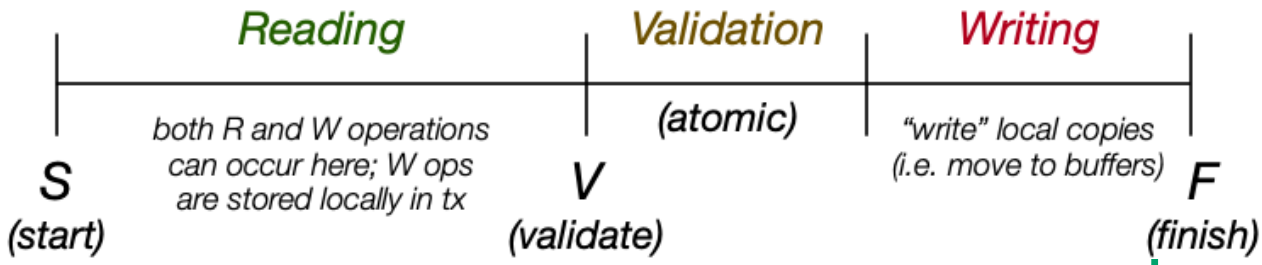
Optimistic concurrency control (OCC) is a strategy to realise this.

❖ Optimistic Concurrency Control (cont)

Under OCC, transactions have three distinct phases:

- Reading: read from database, modify local copies of data
- Validation: check for conflicts in updates
- Writing: commit local copies of data to database

Timestamps are recorded at points *S, V, F* :



❖ Validation

Data structures needed for validation:

- *S* ... set of txs that are reading data and computing results
- *V* ... set of txs that have reached validation (not yet committed)
- *F* ... set of txs that have finished (committed data to storage)
- for each T_i, timestamps for when it reached *S*, *V*, *F*
- $RS(T_i)$ set of all data items read by T_i
- $WS(T_i)$ set of all data items to be written by T_i

Use the *V* timestamps as ordering for transactions

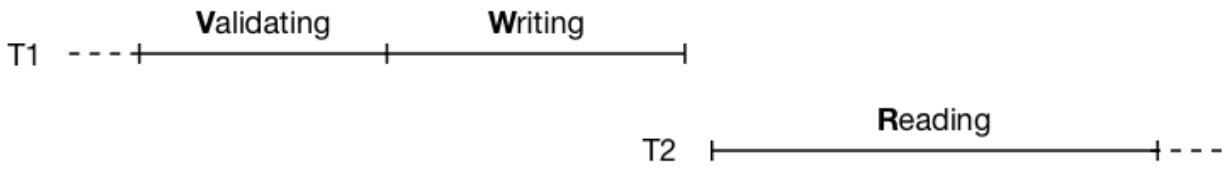
- assume serial tx order based on ordering of $V(T_i)$'s

❖ Validation (cont)

Two-transaction example:

- allow transactions T_1 and T_2 to run without any locking
- check that objects used by T_2 are not being changed by T_1
- if they are, we need to roll back T_2 and retry

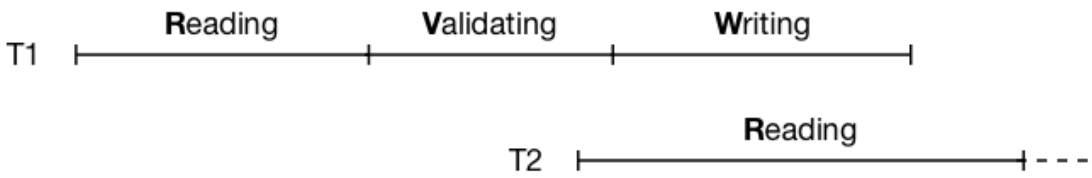
Case 0: serial execution ... no problem



❖ Validation (cont)

Case 1: reading overlaps validation/writing

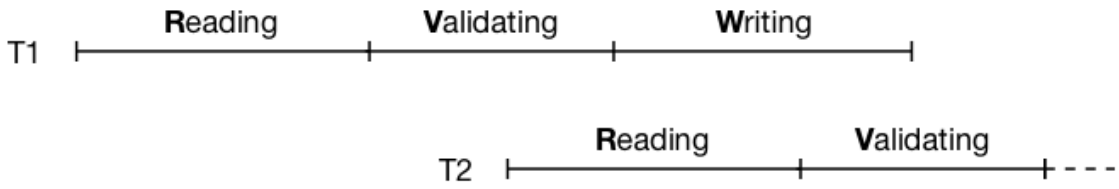
- T_2 starts while T_1 is validating/writing
- if some X being read by T_2 is in $WS(T_1)$
- then T_2 may not have read the updated version of X
- so, T_2 must start again



❖ Validation (cont)

Case 2: reading/validation overlaps validation/writing

- T_2 starts validating while T_1 is validating/writing
- if some X being written by T_2 is in $WS(T_1)$
- then T_2 may end up overwriting T_1's update
- so, T_2 must start again



❖ Validation Check

Validation check for transaction T

- for all transactions $T_i \neq T$
  - if $T \in S$ & $T_i \in F$, then ok
  - if $T \notin V$ & $V(T_i) < S(T) < F(T_i)$,
    then check $WS(T_i) \cap RS(T)$ is empty
  - if $T \in V$ & $V(T_i) < V(T) < F(T_i)$,
    then check $WS(T_i) \cap WS(T)$ is empty

If this check fails for any T_i, then T is rolled back.

❖ Validation Check (cont)

OCC prevents: T reading dirty data, T overwriting T_i's changes

Problems with OCC:

- increased roll backs**
- tendency to roll back "complete" tx's
- cost to maintain *S,V,F* sets

** "Roll back" is relatively cheap

- changes to data are purely local before Writing phase
- no requirement for logging info or undo/redo (see later)

Multi-version Concurrency Control

- Multi-version Concurrency Control
- Concurrency Control in PostgreSQL

❖ 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: $tup_{oldest} \rightarrow tup_{older} \rightarrow tup_{newest}$

When a reader T_i is accessing the database

- ignore any data item D created 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

Some MVCC versions also maintain RTS (TS of last reader)

- don't allow T_i to write D if RTS(D) > TS(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)
- reading an item *V* causes an update of *RTS(V)*
- 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