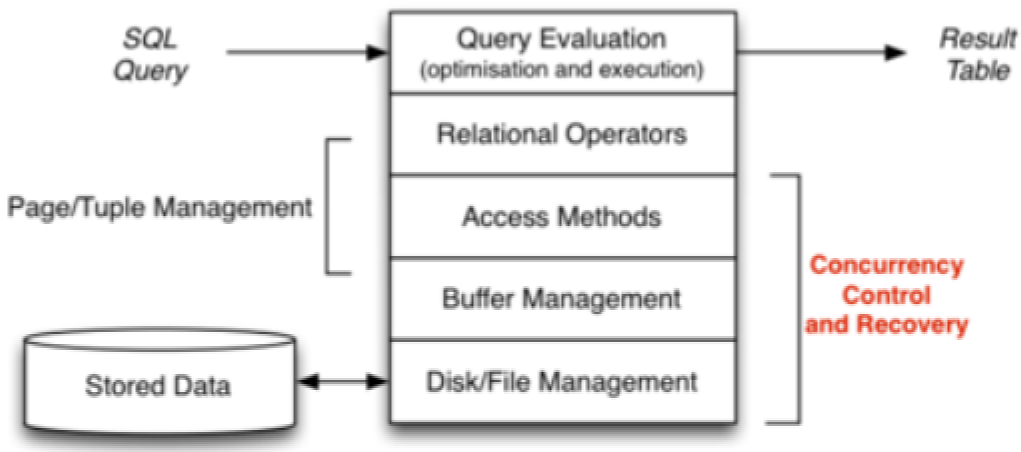
course 11 Transaction Processing

1. Transactions

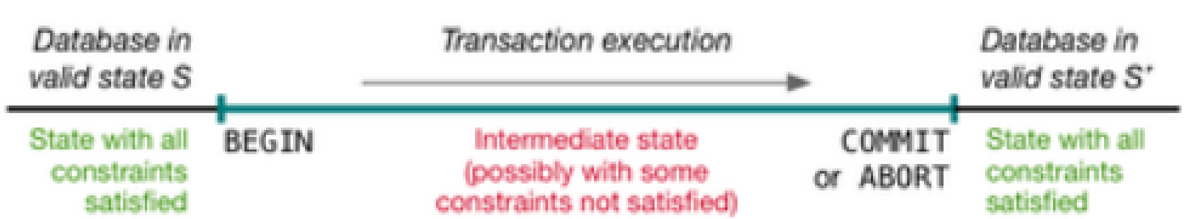
1.1 concurrency in DBMSs



a transaction is **a unit of processing** corresponding to a DB state-change

and in order to achieve satisfactory performance, DBMS allow multiple transaction to execute concurrently

a transaction is a set of operation, like (select, update, insert, …), it is also a **DB state-change operation**



Transactions execute on a collection of data that is

- **shared** - concurrent access by multiple users
- **unstable** - potential for hardware/software failure

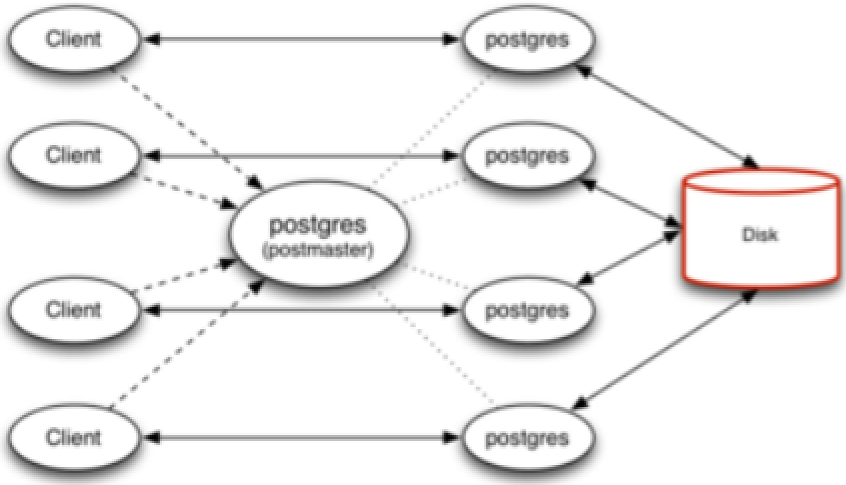
Transactions need an environment that is

- **unshared** - their work is not inadvertantly affected by others
- **stable** - their updates survive even in the face of system failure

when DBMS execute transaction, it should ensure not to happen **“Dead Lock” by:**

- if a transaction commit
  - effects of all operations persist permanently and is visible to all subsequent transactions
- part-way through a transaction
  - other transactions can’t access results of partly-complete computations
- if a transaction abort
  - rolled-back
- if there is a system failure
  - rolled-back
  - database can restored to the consistent state

concurrency in multi-user DBMS like PostgreSQL:

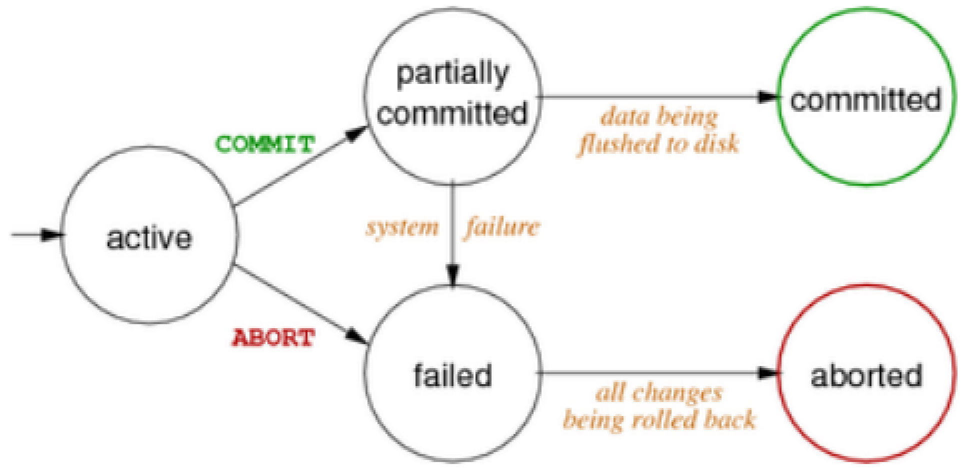


1.2 ACID properties

data security can be ensured if DBMS satisfy the following:

- Atomicity:
  - either all operations of transaction are reflected in database or none are
- Consistency
  - execution of a transaction in isolation perserves data consistency
- Isolation
  - each transaction is unware of other transactions when executing concurrently
- Durability
  - if a transaction commits, its changes persist even after later system failure

Atomicity can be represented by state–transitions:



COMMIT ⇒ all changes preserved, ABORT ⇒ database unchanged

so how to realise ACID properties?

the implementation of Consistency is left to application programmers

we only focus on the **implementation of** Atomicity, Durability, isolation

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

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

Transaction terminology:

to describe transaction effect, we consider:

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

and the relationship between SQL and above operation:

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

the READ, WRITE, ABORT, COMMIT operations occur in the context of some transactions, and involve manipulation of data items X, Y, …

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

1.3 Schedules

a schedules gives sequence of operations that occur, for one transaction, there are only one possible schedules; but for several transactions run concurrently, there are multiple types of schedules.

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

a serial execution of consistent transactions is always consistent but if transactions executunder a cncurrent schedule, the potential exists for conflict among their effects.

1.4 Transaction Anomalies

so what problem occur with concurrent transactions?

The set of phenomena can be characterised broadly under:

- *dirty read*:
  reading data item currently in use by another tx
- *nonrepeateable read*:
  re-reading data item, since changed by another tx
- *phantom read*:
  re-reading result set, since changed by another tx

1.5 Schedule Properties

if a concurrent schedule on a set of transactions's TT

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

and the primary goal of isolation mechanism is:

- arrange execution of individual operations in transaction'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

1.6 transaction failure

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
```

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

1.7 recoverability

Recoverable schedules avoid these kinds of problems.

For a schedule to be recoverable, we require additional constraints

- all tx's T_i that wrote values used by T_j
- must have committed 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.

1.8 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 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.

1.9 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.

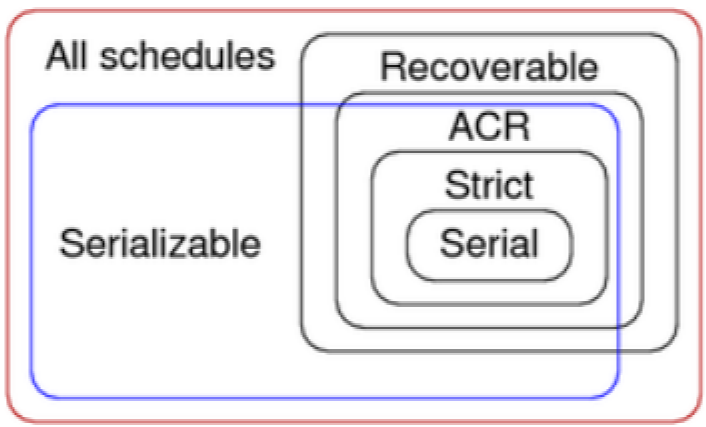
Example: non−strict schedule

```
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)

1.10 classes of schedules



schedules ought to be serializable and strict

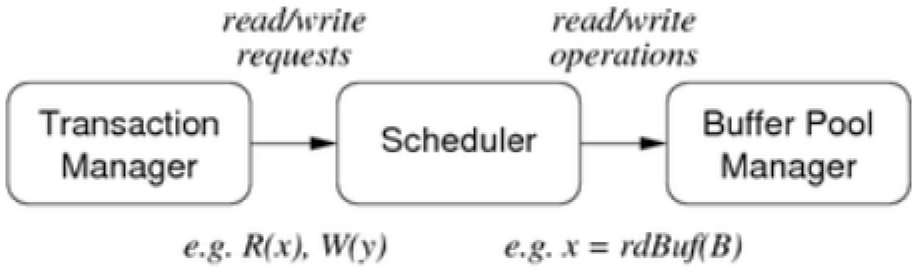
But more serializable/strict → less concurrency

DBMSs allow users to trade off “safety” aganist performance

2. Transaction Isolation

2.1 DBMS Transaction Management

abstract view of DBMS concurrency mechanisms:



the scheduler:

- collects artbitrarily interleaved requests from transaction's
- order their execution to avoid concurrency problems

2.2 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)

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.

2.3 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.

implication of transaction isolation levels:

Isolation level	Dirty read	Norepeatable read	phantom read
read uncommited	possible	possible	possible
read committed	not possible	possible	possible
repeatable read	not possible	not possible	possible

serializable	not possible	not possible	not possible

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

3. Implementing Concurrency Control

3.1 concurrency control

aims of concurrency control schemes:

- avoid transaction anomaly and support as many concurrency as possible

and this scheme should:

- ensure that operations from concurrent transaction occur in a safe order
- when unsafe detectedm need to rollback

Approaches to 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.

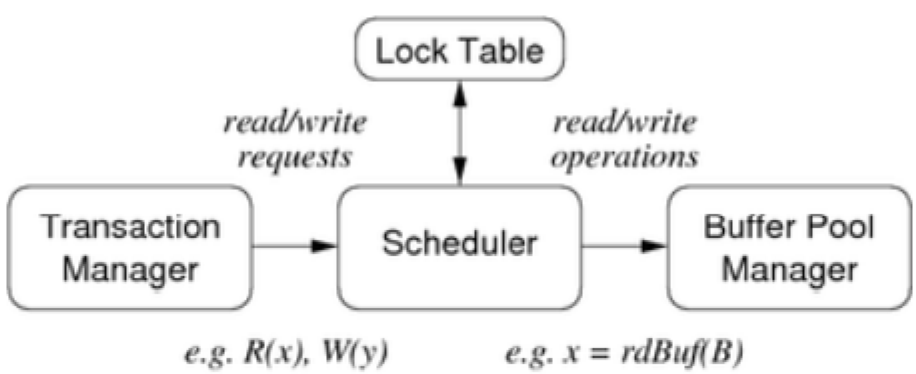
3.2 Lock-based concurrency control

sychronise access to share data item via following rules:

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

blocking causes transaction to wait → reduce concurrency

but also prevents some transaction anomalies



the lock manager: manages the locks requested by the scheduler

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

3.3 Two-Phase Locking

although lock-based concurrency control mechanism do a good job, it can not guarantee serialibility, for this, we require an additional constraint:

- in every transaction, 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 reduces potential concurrency

3.4 Problems with Locking

although appropriate locking can guarantee correctness, however, it also introduces potential undersirable 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.

3.5 Deadlock

deadlock occur when two transactions are waiting for a lock on an item held by the other

so how to deal with deadlock:

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

forcing a transaction to back off can handle deadlock:

- 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

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 $\Rightarrow$ abort T_i
  - *detect* deadlock by periodic check for cycles $\Rightarrow$ 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

properties of deadlock handing 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

3.6 Optimistic concurrency control

although locking do a good job, but it really sacrifice concurrency to earn security

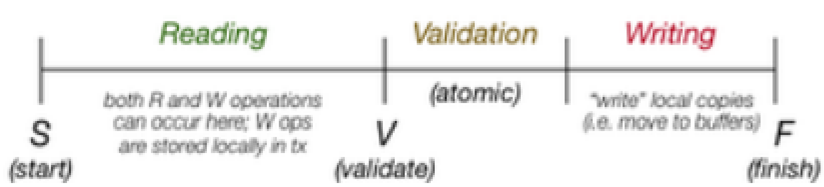
in scenarios where there are far more reads than writes:

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

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:



- in reading phases, both R and W opeartions can occur here, and w ops are stored locally in transaction
- in vaildation phases, do some atomic jobs
- in writing phase, write local copies, like move to buffers

data structures needed for validation:

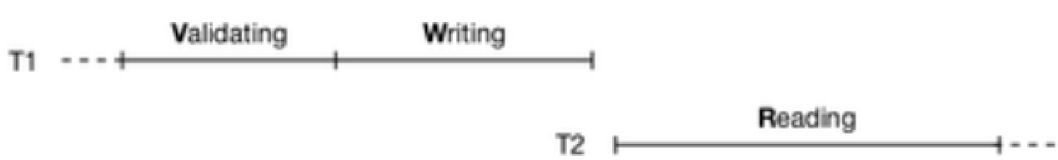
- *S* ... set of txs that are reading data and computing results
- *V* ... set of txs that have reached validation (not yet committed)
- *F* ... set of txs that have finished (committed data to storage)
- for each T_i, timestamps for when it reached *S*, *V*, *F*
- $RS(T_i)$ set of all data items read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

for examples:

Two–transaction example:

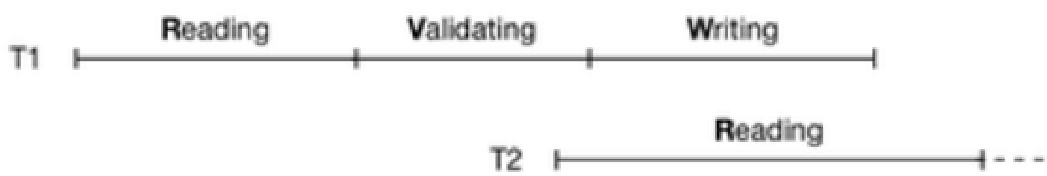
- allow transactions T_1 and T_2 to run without any locking
- check that objects used by T_2 are not being changed by T_1
- if they are, we need to roll back T_2 and retry

Case 0: serial execution ... no problem



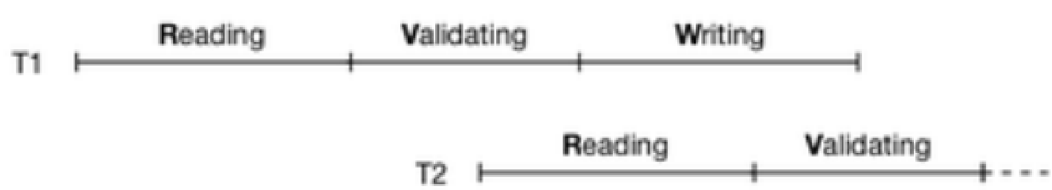
Case 1: reading overlaps validation/writing

- 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



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 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.

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

Problem with OCC:

- increased roll backs
- cost to maintain S,V,F sets

but it can be accpeted, since Roll back is relatively cheap:

- changes to data are stored locally before writing phases
- no requirement for logging info or undo/redo

3.7 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

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)

When a writer T_j attempts to change a data item

- find newest version V satisfying WTS(V) < TS(T_j)

- if no later versions exist, create new version of data item
- if there are later versions, then abort T_j

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

- don't allow T_j to write D if RTS(D) > TS(T_j)

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.

3.8 Concurrency control in PostgreSQL

PostgreSQL uses two styles of concurrency control:

- multi-version concurrency control (MVCC)
    - (used in implementing SQL DML statements **like select**)
- two-phases locking (2PL)
    - (used in DDL statements **like create table**)

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)

implementing a 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

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: `utils/time/tqual.c`

4. Implementing Atomicity/Durability

4.1 Atomicity/Durability

Transactions are *atomic*

- if a tx commits, all of its changes persist in DB
- if a tx aborts, none of its changes occur in DB

Transaction effects are *durable*

- if a tx commits, its effects persist
  (even in the event of subsequent (catastrophic) system failures)

Implementation of atomicity/durability is intertwined.

4.2 Durability

durability begins with a stable disk storage subsystem

and we can prevent/minimise loss/corruption of data due to:

- mem/disk transfer corruption ⇒ parity checking
- sector failure ⇒ mark "bad" blocks
- disk failure ⇒ RAID (levels 4,5,6)
- destruction of computer system ⇒ off-site backups

4.3 Dealing with transactions

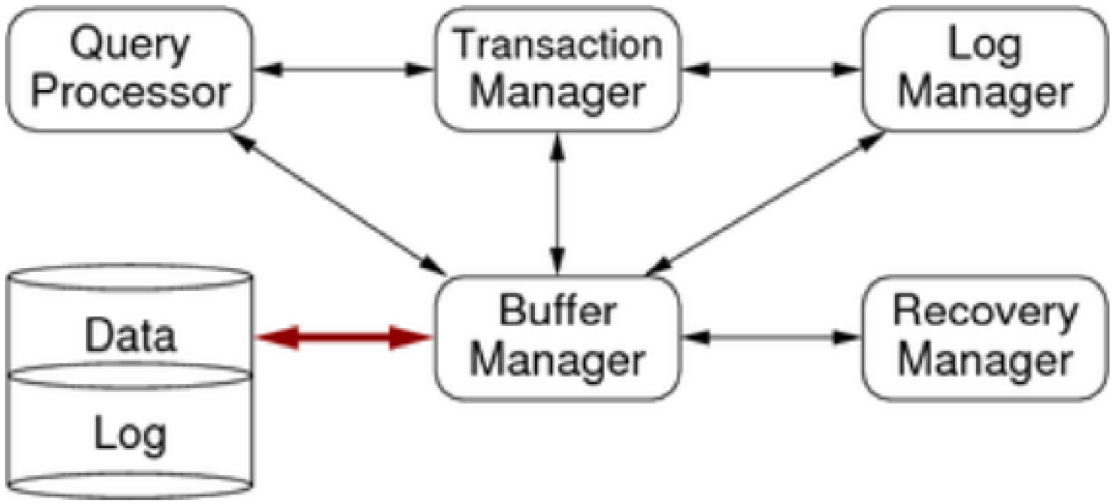
the remaining failure modes that we need to consider:

- failure of DBMS processes or operating system
- failure of transactions (ABORT)

standard technique for managing these:

- keep a log of changes made to database
- use this log to restore state in case of failures

4.4 Architecture for Atomicity/Durability



4.5 Execution of Transactions

transactions dela with three address spaces:

- stored data on the disk (representing global DB state)
- data in memory buffers (where held for sharing by tx's)
- data in their own local variables (where manipulated)

each of these may hold a different version of a DB object

becaues PostgreSQL processes make heavy use of shared buffer pool → transactions do not deal with much local data

Operations available for data transfer:

- `INPUT(X)` ... read page containing `X` into a buffer
- `READ(X,v)` ... copy value of `X` from buffer to local var `v`
- `WRITE(X,v)` ... copy value of local var `v` to `X` in buffer
- `OUTPUT(X)` ... write buffer containing `X` to disk

`READ/WRITE` are issued by transaction.

`INPUT/OUTPUT` are issued by buffer manager (and log manager).

`INPUT/OUTPUT` correspond to `getPage()`/`putPage()` mentioned above

examples of transaction execution:

```
-- implements A = A*2; B = B+1;
BEGIN
READ(A,v); v = v*2; WRITE(A,v);
READ(B,v); v = v+1; WRITE(B,v);
COMMIT
```

READ accesses the buffer manager and may cause INPUT

COMMIT needs to ensure that buffer contents go to disk

States as the transaction executes:

t	Action	v	Buf(A)	Buf(B)	Disk(A)	Disk(B)
(0)	BEGIN	.	.	.	8	5
(1)	READ(A,v)	8	8	.	8	5
(2)	v = v*2	16	8	.	8	5
(3)	WRITE(A,v)	16	16	.	8	5
(4)	READ(B,v)	5	16	5	8	5
(5)	v = v+1	6	16	5	8	5
(6)	WRITE(B,v)	6	16	6	8	5
(7)	OUTPUT(A)	6	16	6	16	5
(8)	OUTPUT(B)	6	16	6	16	6

After transaction completes, we must have either Disk(A)=8, DIsk(B)=5

or Disk(A)=16, Disk(B)=6

if system crashes before 8, may need to undo disk changes

if system crashes after 8, may need to redo disk changes

4.6 Transactions and Buffer Pool

two issues arise:

- forcing: OUTPUT buffer on each WRITE
    - it can ensure durability, disk always consistent with buffer pool
    - but it will bring poor performance, which defeats purpose of having buffer pool
- stealing: replace buffers of uncommitted transactions
    - if we don’t, poor throughput (transaction was blocked on buffers)
    - if we do, seems to cause atomicity problems?

Handling stealing:

- transaction T loads page P and makes changes
- T_2 needs a buffer, and P is the "victim"
- P is output to disk (it's dirty) and replaced
- if T aborts, some of its changes are already "committed"
- must log values changed by T in P at "steal–time"
- use these to UNDO changes in case of failure of T

Handling no forcing:

- transaction T makes changes & commits, then system crashes
- but what if modified page P has not yet been output?
- must log values changed by T in P as soon as they change
- use these to support REDO to restore changes

4.7 Logging

there are three styles of logging:

- undo:
- redo:
- undo/redo:

(write-ahead logging) logging requires:

- a sequential file of log records
- each log record describes a change to a data item
- **log records are written first**
- **actual changes to data are written later**

PostgreSQL uses WAL

4.8 Undo logging

Undo logging is a simple form of logging which ensures atomicity

Log file consists of a sequence of small records:

- `<START T>` ... transaction `T` begins
- `<COMMIT T>` ... transaction `T` completes successfully
- `<ABORT T>` ... transaction `T` fails (no changes)
- `<T,X,v>` ... transaction `T` changed value of `X` from `v`

Notes:

- we refer to `<T,X,v>` generically as `<UPDATE>` log records
- update log entry created for each `WRITE` (not `OUTPUT`)
- update log entry contains *old* value (new value is not recorded)

Data must be written to disk in the following order:

1. `<START>` transaction log record
2. `<UPDATE>` log records indicating changes
3. the changed data elements themselves
4. `<COMMIT>` log record

For example transaction, we could get:

t	Action	v	B(A)	B(B)	D(A)	D(B)	Log
(0)	BEGIN	.	.	.	8	5	<START T>
(1)	READ(A,v)	8	8	.	8	5	
(2)	v = v*2	16	8	.	8	5	
(3)	WRITE(A,v)	16	16	.	8	5	<T,A,8>
(4)	READ(B,v)	5	16	5	8	5	
(5)	v = v+1	6	16	5	8	5	
(6)	WRITE(B,v)	6	16	6	8	5	<T,B,5>
(7)	FlushLog						
(8)	StartCommit						
(9)	OUTPUT(A)	6	16	6	16	5	
(10)	OUTPUT(B)	6	16	6	16	6	
(11)	EndCommit						<COMMIT T>
(12)	FlushLog						

Notes: T is not regarded as committed until (12) completes

Simplified view of recovery using UNDO logging:

- scan *backwards* through log
  - if `<COMMIT T>`, mark `T` as committed
  - if `<T,X,v>` and `T` not committed, set `X` to `v` on disk
  - if `<START T>` and `T` not committed, put `<ABORT T>` in log

Assumes we scan entire log; use checkpoints to limit scan.

4.9 Checkpointing

what is Checkpoint

- all of log prior to checkpoint can be ignored for recovery

problems：there are many concurrent/overlapping transactions

how to know that all have finished?

1. periodically, write log record `<CHKPT (T1,..,Tk)>`
   (contains references to all active transactions ⇒ active tx table)
2. continue normal processing (e.g. new tx's can start)
3. when all of `T1,..,Tk` have completed,
   write log record `<ENDCHKPT>` and flush log

Recovery: scan backwards through log file processing as before

Determining where to stop depends on ...

- whether we meet `<ENDCHKPT>` or `<CHKPT...>` first

If we encounter `<ENDCHKPT>` first:

- we know that all incomplete tx's come after prev `<CHKPT...>`
- thus, can stop backward scan when we reach `<CHKPT...>`

If we encounter `<CHKPT (T1,..,Tk)>` first:

- crash occurred *during* the checkpoint period
- any of `T1,..,Tk` that committed before crash are ok
- for uncommitted tx's, need to continue backward scan

4.10 Redo Logging

problems with UNDO logging:

- all changed data must be output to disk before committing
- conflicts with optimal use of the buffer pool

alternative approach is redo logging:

- allow changes to remain only in buffers after commit
- write records to indicate what changes are “pending”

- after a crash, can apply changes during recovery

Simplified view of recovery using REDO logging:

- identify all committed tx's (backwards scan)
- scan *forwards* through log
  - if `<T,X,v>` and `T` is committed, set `X` to `v` on disk
  - if `<START T>` and `T` not committed, put `<ABORT T>` in log

Assumes we scan entire log; use checkpoints to limit scan.

4.11 Undo/Redo Logging

UNDO logging and REDO logging are incompatible in

- order of outputting `<COMMIT T>` and changed data
- how data in buffers is handled during checkpoints

Undo/Redo logging combines aspects of both

- requires new kind of update log record
  `<T,X,v,v'>` gives both old and new values for `X`
- removes incompatibilities between output orders

As for previous cases, requires write–ahead of log records.

Undo/redo loging is common in practice; Aries algorithm.

4.12 Recovery in PostgreSQL

PostgreSQL uses write-ahead undo/redo style logging

it also uses multi-version concurrency control, which tags each record with a transaction and update timestamp

MVCC simplifies some aspects of undo/redo, e.g.

- some info required by logging is already held in each tuple
- no need to undo effects of aborted transacitons, just use old version