Week 12

Transaction Processing

1/66

Transaction (tx) = application-level atomic op, multiple DB ops

Concurrent transactions are

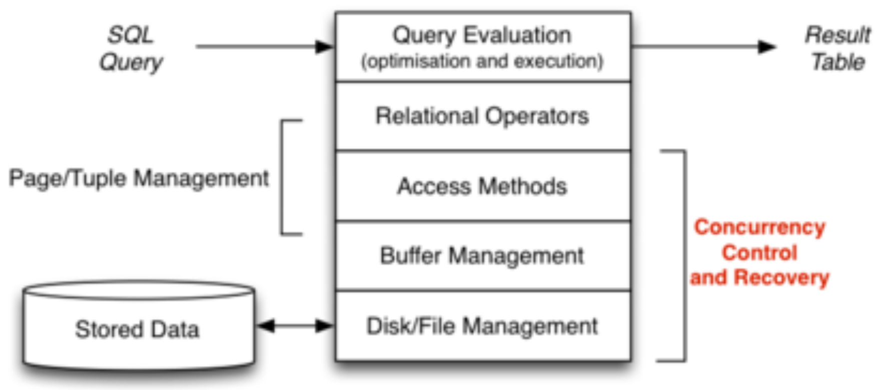
- desirable, for improved performance
- problematic, because of potential unwanted interactions

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

2/66



Transaction Isolation

Transaction Isolation

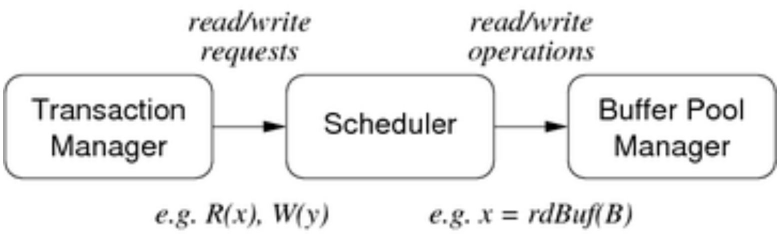
4/66

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

5/66

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

6/66

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 1: Serializability Checking

7/66

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)
```

Transaction Isolation Levels

8/66

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

9/66

Meaning 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

10/66

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

... Transaction Isolation Levels

11/66

A PostgreSQL tx consists of a sequence of SQL statements:

BEGIN S_1; S_2; ... S_n; 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

Implementing Concurrency Control

Concurrency Control

13/66

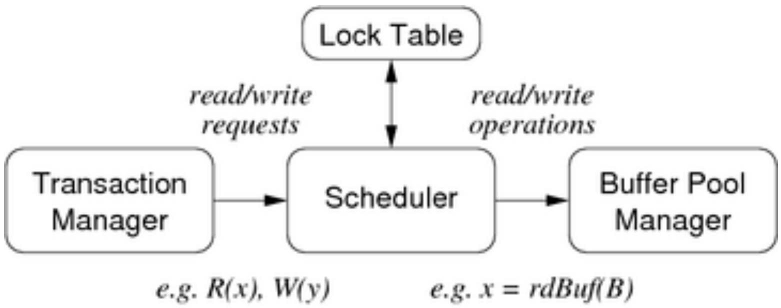
Approaches to concurrency control:

- *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.

Lock-based Concurrency Control

14/66

Locks introduce additional mechanisms in DBMS:



The Lock Manager

- manages the locks requested by the scheduler

... Lock-based Concurrency Control

15/66

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

16/66

Synchronise access to shared data items via following 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 an 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

17/66

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

18/66

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

19/66

Appropriate locking can guarantee correctness.

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

20/66

Deadlock occurs when two transactions are waiting for a lock on an item held by the other.

Example:

```
T1: Lw(A) R(A)            Lw(B) ......
T2:            Lw(B) R(B)       Lw(A) .....
```

How to deal with deadlock?

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

... Deadlock

21/66

Handling deadlock involves forcing a transaction to "back off".

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

---

... Deadlock

22/66

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

---

... Deadlock

23/66

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

Exercise 2: Deadlock Handling

24/66

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 wait-for graph for the locks evolves.

Show how any deadlocks might be resolved via this graph.

Optimistic Concurrency Control

25/66

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

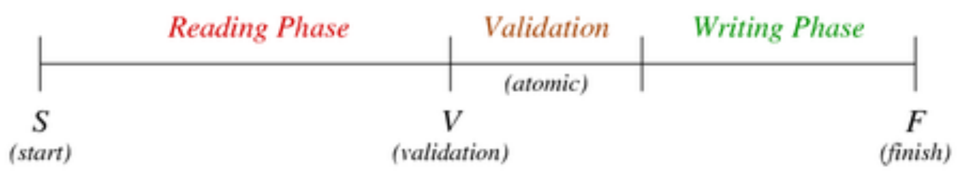
... Optimistic Concurrency Control

26/66

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 noted:



... Optimistic Concurrency Control

27/66

Data structures needed for validation:

- *A* ... 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 *A*, *V*, *F*
- $R(T_i)$ set of all data items read by T_i
- $W(T_i)$ 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

... Optimistic Concurrency Control

28/66

Validation check for transaction T

- for all transactions $T_i \neq T$
  - if $V(T_i) < A(T) < F(T_i)$, then check $W(T_i) \cap R(T)$ is empty
  - if $V(T_i) < V(T) < F(T_i)$, then check $W(T_i) \cap W(T)$ is empty

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

Prevents: T reading dirty data, T overwriting T_i's changes

Problems: rolls back "complete" tx's, cost to maintain A,V,F sets

---

Multi-version Concurrency Control

29/66

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

30/66

WTS = timestamp of last writer; RTS = timestamp of last reader

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

When a reader T_i is accessing the database

- ignore any data item created after T_i started (WTS > TS(T_i))
- use only newest version V satisfying 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 RTS(V) ≤ TS(T_j), create new version of data item
- if RTS(V) > TS(T_j), reject the write and abort T_j

... Multi-version Concurrency Control

31/66

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

32/66

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

33/66

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

34/66

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

35/66

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

36/66

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`**

... Concurrency Control in PostgreSQL

37/66

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

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

E.g. T_1 does select, then concurrent T_2 deletes some of T_1's selected tuples

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

E.g. T_1 counts tuples while T_2 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

Exercise 3: Locking in PostgreSQL

38/66

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;
```

Implementing Atomicity/Durability

Atomicity/Durability

40/66

Reminder:

Transactions are *atomic*

- if a tx commits, all of its changes occur 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.

Durability

41/66

What kinds of "system failures" do we need to deal with?

- single-bit inversion during transfer mem-to-disk
- decay of storage medium on disk (some data changed)
- failure of entire disk device (no longer accessible)
- failure of DBMS processes (e.g. `postgres` crashes)
- operating system crash, power failure to computer room
- complete destruction of computer system running DBMS

The last requires off-site *backup*; all others should be locally recoverable.

... Durability

42/66

Consider following scenario:

Desired behaviour after system restart:

- all effects of T1, T2 persist
- as if T3, T4 were aborted (no effects remain)

... Durability

43/66

Durabilty begins with a *stable disk storage subsystem*.

i.e. effects of `putPage()` and `getPage()` are consistent

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

Dealing with Transactions

44/66

The remaining "failure modes" that we need to consider:

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

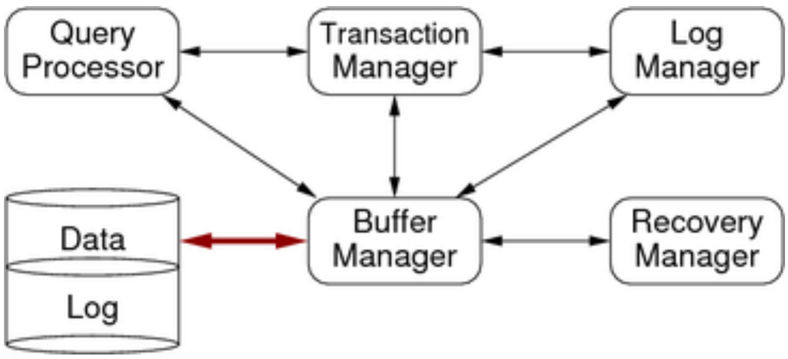
Standard technique for managing these:

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

Architecture for Atomicity/Durability

45/66

How does a DBMS provide for atomicity/durability?



Execution of Transactions

46/66

Transactions deal with three address spaces:

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

Each of these may hold a different "version" of a DB object.

PostgreSQL processes share buffer pool ⇒ not much local data.

... Execution of Transactions

47/66

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

... Execution of Transactions

48/66

Example 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.

... Execution of Transactions

49/66

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 tx 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.

Transactions and Buffer Pool

50/66

Two issues arise w.r.t. buffers:

- *forcing* ... `OUTPUT` buffer on each `WRITE`

    - ensures durability; disk always consistent with buffer pool
    - poor performance; defeats purpose of having buffer pool
- *stealing* ... replace buffers of uncommitted tx's
    - if we don't, poor throughput (tx's blocked on buffers)
    - if we do, seems to cause atomicity problems?

Ideally, we want stealing and not forcing.

... Transactions and Buffer Pool

51/66

Handling *stealing*:

- page P, held by tx T, is output to disk and replaced
- if T aborts, some of its changes are already "committed"
- must log changed values in P at "steal-time"
- use these to UNDO changes in case of failure of T

Handling *no forcing*:

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

Logging

52/66

Three "styles" of logging

- *undo* ... removes changes by any uncommitted tx's
- *redo* ... repeats changes by any committed tx's
- *undo/redo* ... combines aspects of both

All approaches require:

- 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

Known as *write-ahead logging*

Undo Logging

53/66

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)

... Undo Logging

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

Note: sufficient to have `<T,X,v>` output before `X`, for each `X`

... Undo Logging

For the example transaction, we would 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
```

Note that T is not regarded as committed until (11).

... Undo Logging

Simplified view of recovery using UNDO logging:

```
committedTrans = abortedTrans = startedTrans = {}
for each log record from most recent to oldest {
    switch (log record) {
    <COMMIT T> : add T to committedTrans
    <ABORT T>  : add T to abortedTrans
    <START T>  : add T to startedTrans
    <T,X,v>    : if (T in committedTrans)
                     // don't undo committed changes
                 else  // roll-back changes
                     { WRITE(X,v); OUTPUT(X) }
}   }
for each T in startedTrans {
    if (T in committedTrans) ignore
    else if (T in abortedTrans) ignore
    else write <ABORT T> to log
}
flush log
```

Checkpointing

57/66

Simple view of recovery implies reading entire log file.

Since log file grows without bound, this is infeasible.

Eventually we can delete "old" section of log.

- i.e. where *all* prior transactions have committed

This point is called a *checkpoint*.

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

... Checkpointing

58/66

Problem: 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

Note: tx manager maintains chkpt and active tx information

... Checkpointing

59/66

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

Redo Logging

60/66

Problem 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

... Redo Logging

61/66

Requirement for redo logging: *write-ahead rule*.

Data must be written to disk as follows:

1. start transaction log record
2. update log records indicating changes
3. then commit log record (`OUTPUT`)
4. then `OUTPUT` changed data elements themselves

Note that update log records now contain `<T,X,v'>`,
where `v'` is the *new* value for `X`.

... Redo Logging

62/66

For the example transaction, we would 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,16>
(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,6>
(7)  COMMIT                                    <COMMIT T>
(8)  FlushLog
(9)  OUTPUT(A)      6   16     6    16     5
(10) OUTPUT(B)      6   16     6    16     6
```

Note that T is regarded as committed as soon as (8) completes.

Undo/Redo Logging

63/66

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.

... Undo/Redo Logging

64/66

For the example transaction, we might 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,8,16>
(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,5,6>
(7)  FlushLog
(8)  StartCommit
(9)  OUTPUT(A)      6   16     6    16     5
(10)                                           <COMMIT T>
(11) OUTPUT(B)      6   16     6    16     6
```

Note that T is regarded as committed as soon as (10) completes.

Recovery in PostgreSQL

65/66

PostgreSQL uses write-ahead undo/redo style logging.

It also uses multi-version concurrency control, which

- tags each record with a tx 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 tx's; use old version

... Recovery in PostgreSQL

66/66

Transaction/logging code is distributed throughout backend.

Core transaction code is in **`src/backend/access/transam`**.

Transaction/logging data is written to files in **`PGDATA/pg_xlog`**

- a number of very large files containing log records
- old files are removed once all txs noted there are completed
- new files added when existing files reach their capacity (16MB)
- number of tx log files varies depending on tx activity

Produced: 24 May 2016