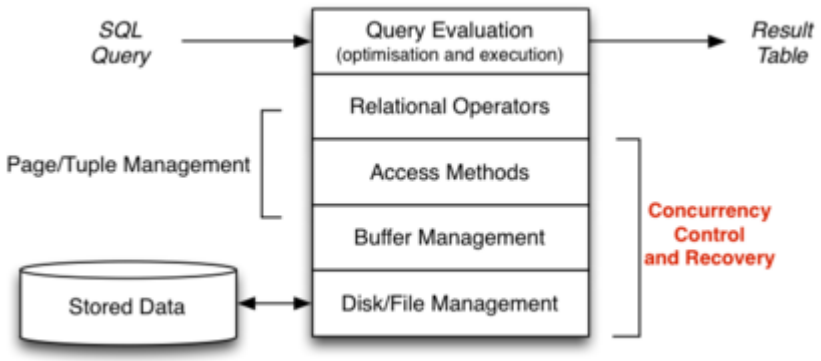
Transaction Processing

Transaction Processing

Where transaction processing fits in the DBMS:



Transactions

A *transaction* is:

- a *unit of processing* corresponding to a DB state–change

Transactions occur naturally in context of DB applications, e.g.

- booking an airline or concert ticket
- transferring funds between bank accounts
- updating stock levels via point–of–sale terminal
- enrolling in a course or class

In order to achieve satisfactory performance (throughput):

- DBMSs allow multiple transactions to execute concurrently

(Notation: we use the abbreviation "**tx**" as a synonym for "transaction")

... Transactions

A *transaction*

- represents an *operational unit* at the application level
- but typically comprises *multiple operations* in the DBMS

E.g. select ... update ... insert ... select ... insert ...

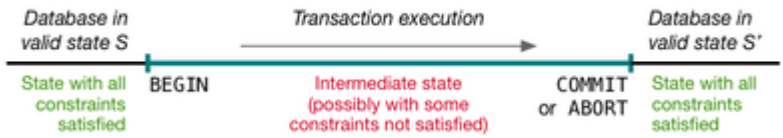
A transaction can end in two possible ways:

- *commit* ... effects of **all** DBMS operations in tx are visible
- *abort* ... effects of **no** DBMS operations in tx are visible

Above is essentially the *atomicity* requirement in *A*CID (see later).

... Transactions

A transaction is a DB state-change operation.



Assume that the code of the transaction

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

Above is essentially the *consistency* requirement in A*C*ID (see later).

... Transactions 5/153

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

Goal: data integrity should be maintained at all times (for each tx)

... Transactions 6/153

If a transaction commits, must ensure that

- effects of all operations persist permanently
- changes are visible to all subsequent transactions

Part-way through a transaction, must ensure that

- other txs don't see results of partly-complete computations

If a transaction aborts, must ensure that

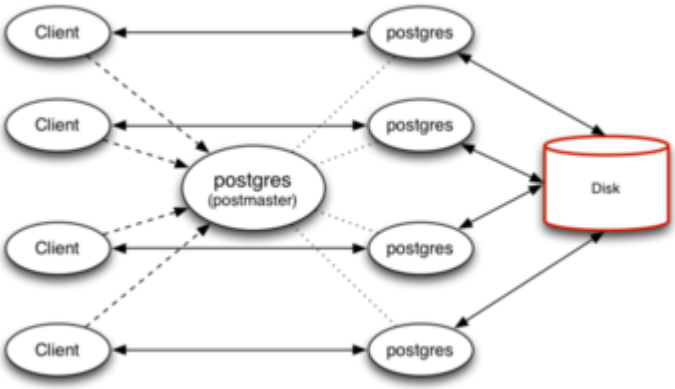
- effects of any DB operations are "cleaned up" (rolled-back)

If there is a system failure, must ensure that on restart

- the database is restored to a consistent state
- all partly-complete transactions are rolled-back

Concurrency in DBMSs 7/153

Concurrency in a multi-user DBMS like PostgreSQL:



ACID Properties

Data integrity is assured if transactions satisfy the following:

Atomicity

- Either all operations of a transaction are reflected in database or none are.

Consistency

- Execution of a transaction in isolation preserves data consistency.

Isolation

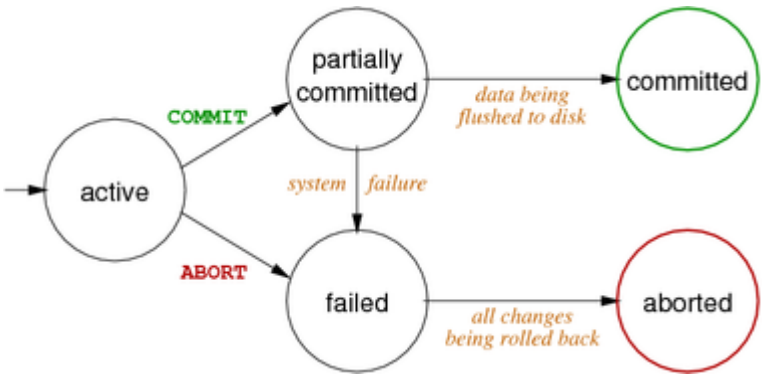
- Each transaction is "unaware" of other transactions executing concurrently.

Durability

- If a transaction commits, its changes persist even after later system failure.

... ACID Properties

Atomicity can be represented by state–transitions:



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

... ACID Properties

Transaction processing:

- the study of techniques for realising ACID properties

Consistency is the property mentioned earlier:

- 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 thus focusses on: Atomicity, Durability, Isolation

---

... ACID Properties 11/153

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)

Review of Transaction Terminology 12/153

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

... Review of Transaction Terminology 13/153

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 stored procedures (e.g. PLpgSQL)

... Review of Transaction Terminology 14/153

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 that occur

- when a set of tx's T_1 .. T_n run concurrently
- operations from invidual T_i's are interleaved

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

For a single tx, there is a single schedule (its operations in order).

For a group of tx's, there are *very many* possible schedules.

... Schedules

Consider a simple banking transaction, expressed in "PLpgSQL":

```
create function
   withdraw(Acct integer, Required float) returns text
as $$
declare
   Bal float;
begin
   select balance into Bal from Accounts where id=Acct;  R
   Bal := Bal - Required;
   update Accounts set balance=Bal where id=Acct;        W
   if (Bal < 0) then
      rollback; return 'Insufficient funds';             A
   else
      commit; return 'New balance: '||Bal::text;         C
   end if;
end;
$$ language plpgsql;
```

Notes:

- a better way to implement this would be to check before updating
- you can't embed tx-type commands inside PLpgSQL functions
- the `begin` at the start of the function does not begin a tx

... Schedules

If tx T = `withdraw(`A`,`R`)` it has two possible schedules

- Fail: $R_T(A)$ $W_T(A)$ A_T
- OK: $R_T(A)$ $W_T(A)$ C_T

If tx T = `withdraw(`A`,`$R1$`)` and tx S = `withdraw(`B`,`$R2$`)`
some possible schedules are:

- $R_T(A)$ $W_T(A)$ A_T $R_S(B)$ $W_S(B)$ C_S (serial schedule)
- $R_T(A)$ $W_T(A)$ $R_S(B)$ $W_S(B)$ A_T C_S
- $R_T(A)$ $R_S(B)$ $W_S(B)$ $W_T(A)$ A_T C_S
- $R_S(B)$ $W_S(B)$ C_S $R_T(A)$ $W_T(A)$ A_T (serial schedule)

... Schedules 18/153

Serial schedules have no interleave of operations from different tx's.

Why serial schedules are good:

- each transaction is correct (consistency)
  (leaves the database in a consistent state if run to completion individually)
- the database starts in a consistent state
- the first transaction completes, leaving the DB consistent
- the next transaction completes, leaving the DB consistent

As would occur e.g. in a single-user database system.

... Schedules 19/153

With different-ordered serial executions, tx's may get different results.

I.e. $StateAfter(T_1;T_2) = StateAfter(T_2;T_1)$ is not generally true.

Consider the following two transactions:

```
T1 : select sum(salary)
     from Employee where dept='Sales'

T2 : insert into Employee
     values (....,'Sales',...)
```

If we execute T_1 then T_2 we get a smaller salary total than if we execute T_2 then T_1.

In both cases, however, the salary total is *consistent* with the state of the database *at the time* the transaction is executed.

... Schedules 20/153

A serial execution of consistent transactions is always consistent.

If transactions execute under a concurrent (nonserial) schedule, the potential exists for conflict among their effects.

In the worst case, the effect of executing the transactions ...

- is to leave the database in an inconsistent state
- even though each transaction, by itself, *is* consistent

So why don't we observe such problems in real DBMSs? ...

- *concurrency control* mechanisms handle them (see later).

Not all concurrent executions cause problems.

For example, the schedules

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

or

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

or ...

leave the database in a consistent state.

Example Transaction #1

Problem: Allocate a seat on a plane flight

Implement as a function returning success status:

```
function allocSeat(paxID    integer,
                   flightID integer,
                   seatNum  string)
         returning boolean
{
    check whether seat currently occupied
    if (already occupied)
        tell pax seat taken; return !ok
    else
        assign pax to seat; return ok
}
```

Assume schema:

```
Flight(flightID, flightNum, flightTime, ...)
SeatingAlloc(flightID, seatNum, paxID, ...)
```

PLpgSQL implementation for seat allocation:

```
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;
```

If customer Cust1 executes allocSeat(Cust1,f,23B)

- Cust1 sees that seat 23B is available
- Cust1 allocates seat 23B for themselves

If customer Cust2 then executes allocSeat(Cust2,f,23B)

- Cust2 sees that seat 23B is already booked
- allocSeat() returns with failure; no change to DB

The system behaves as required ⇒ tx is consistent.

Consider two customers trying allocSeat(?,f,23B) simultaneously.

A possible order of operations ...

- Cust1 sees that seat 23B is available
- Cust2 sees that seat 23B is available
- Cust1 allocates seat 23B for themselves
- Cust2 allocates seat 23B for themselves

Cause of problem: unfortunate interleaving of operations within concurrent transactions.

Serial execution (e.g. enforced by locks) could solve these kinds of problem.

Example Transaction #2

Problem: transfer funds between two accounts in same bank.

Implement as a function returning success status:

```
function transfer(sourceAcct integer,
                  destAcct   integer,
                  amount     real)
         returning boolean
{
    check whether sourceAcct is valid
    check whether destAcct is valid
    check whether sourceAcct has
          sufficient funds (>= amount)
    if (all ok) {
        withdraw money from sourceAcct
        deposit money into destAcct
}   }
```

PLpgSQL for funds transfer between accounts:

```
create or replace function
    transfer(source int, dest int, amount float)
    returns boolean
as $$
declare
    ok   boolean := true;
    acct Accounts%rowtype;
```

```
begin
    select * into acct
    from   Accounts where id=source;
    if (not found) then
        raise warning 'Invalid Withdrawal Account';
        ok := false;
    end if;
    select * from Accounts where id=dest;
    if (not found) then
        raise warning 'Invalid Deposit Account';
        ok := false;
    end if;
    if (acct.balance < amount) then
        raise warning 'Insufficient funds';
        ok := false;
    end if;
    if (not ok) then rollback; return false; end if;
    update Accounts
    set    balance := balance - amount
    where  id = source;
    update Accounts
    set    balance := balance + amount
    where  id = dest;
    commit;
    return true;
end;
$$ language plpgsql;
```

... Example Transaction #2

If customer transfers $1000 from Acct1 to Acct2

- Acct1 and Acct2 both exist; Acct1 has > $1000
- remove $1000 from Acct1; add $1000 to Acct2

But if Cust1 and Cust2 both transfer from Acct1 together

- all accounts exist, and Acct1 contains $1500
- Cust1 checks Acct1; has sufficient funds
- Cust2 checks Acct1; has sufficient funds
- Cust1 removes money from Acct1; adds to Acct2
- Cust2 removes money from Acct1; adds to Acct2

But account ran out of money after Cust1 took their cash.

Similar to earlier problem; could be fixed by serialization.

... Example Transaction #2

Consider customer transfers $1000 from Acct1 to Acct2

- Acct1 and Acct2 both exist; Acct1 has > $1000
- remove $1000 from Acct1; then system failure

If transactions are not atomic/durable:

- money removed from Acct1, no money added to Acct2
- customer is not happy

Transaction Anomalies

What problems can 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

... Transaction Anomalies

Dirty read: a transaction reads data written by a concurrent uncommitted transaction

Example:

```
     Transaction T1        Transaction T2
(1)  select a into X
     from R where id=1
(2)                        select a into Y
                           from R where id=1
(3)  update R set a=X+1
     where id=1
(4)  commit
(5)                        update R set a=Y+1
                           where id=1
(6)                        commit
```

Effect: T1's update on `R.a` is lost.

... Transaction Anomalies

Nonrepeatable read: a transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read)

Example:

```
     Transaction T1    Transaction T2
(1)  select * from R
     where id=5
(2)                    update R set a=8
                       where id=5
(3)                    commit
(4)  select * from R
     where id=5
```

Effect: T1 runs same query twice; sees different data

... Transaction Anomalies

Phantom read: a transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction

Example:

```
     Transaction T1    Transaction T2
(1)  select count(*)
     from R where a=5
(2)                    insert into R(id,a,b)
                              values (2,5,8)
(3)                    commit
(4)  select count(*)
     from R where a=5
```

Effect: T1 runs same query twice; sees different result set

Example of Transaction Failure

The above examples generally assumed that transactions committed.

Additional problems can arise when transactions abort.

We give examples using the following two transactions:

```
T1: read(X)           T2: read(X)
    X := X + N            X := X + M
    write(X)              write(X)
    read(Y)
    Y := Y - N
    write(Y)
```

and initial DB state `X=100`, `Y=50`, `N=5`, `M=8`.

... Example of Transaction Failure

Consider the following schedule where one transaction fails:

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

Transaction `T1` aborts after writing `X`.

The abort *will* rollback the changes to `X`, but where the undo occurs can affect the results.

Consider three places where rollback might occur:

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

Transaction Failure – Case 1

This scenario is ok. `T1`'s effects have been eliminated.

```
Database   Transaction T1        Transaction T2
---------  ------------------    --------------
X    Y                X    Y                X
100  50               ?    ?                ?
           read(X)    100
           X:=X+N     105
105        write(X)
           abort
100        rollback
                                 read(X)    100
                                 X:=X+M     108
108                              write(X)
---------
108  50
```

Transaction Failure – Case 2

In this scenario, some of `T1`'s effects have been retained.

```
Database   Transaction T1        Transaction T2
---------  ------------------    --------------
X    Y                X    Y                X
```

```
100  50              ?    ?               ?
          read(X)    100
          X:=X+N     105
105       write(X)
          abort
                                read(X)    105
                                X:=X+M     113
100       rollback
113                               write(X)
---------
113  50
```

Transaction Failure – Case 3

In this scenario, `T2`'s effects have been lost, even after commit.

```
Database   Transaction T1        Transaction T2
---------  ------------------    --------------
X    Y               X    Y               X
100  50              ?    ?               ?
          read(X)    100
          X:=X+N     105
105       write(X)
          abort
                                read(X)    105
                                X:=X+M     113
113                               write(X)
100       rollback
---------
100  50
```

Transaction Isolation

Transaction Isolation

If transactions always run "single–user":

- they are easier to code
  (programmers concentrate on getting application semantics correct)
- there is no danger of "interference"
  (correct transactions won't interact to produce an incorrect result)

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

- transaction T_1 maps DB from valid state D_0 to valid state D_1
- transaction T_2 maps DB from valid state D_1 to valid state D_2
- etc. etc.

... Transaction Isolation

In practice, serial execution gives poor performance.

We need approaches that allow "safe" concurrency

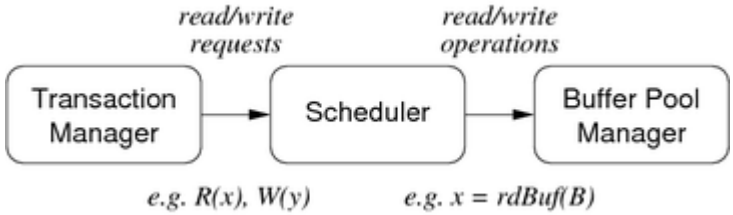
- to improve overall system performance (throughput)
- to avoid the concurrency problems mentioned earlier

The remainder of this discussion involves

- what, exactly, do we mean by "safe" concurrency?
- DBMS mechanisms that can achieve it in practice

DBMS Transaction Management

Abstract view of DBMS concurrency mechanisms:



The Scheduler

- collects arbitrarily interleaved requests from tx's
- orders their execution to avoid concurrency problems

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 different interleaving of *R/W* operations

S_1 and S_2 are *equivalent* if

- $StateAfter(S_1) = StateAfter(S_2)$
  (i.e. the final state yielded by S_1 is the same as the final state yielded by S_2)

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

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

Under these circumstances, consistency is guaranteed.

... Serializability

Two formulations of serializability:

- *conflict serializibility*
  - i.e. conflicting read/write operations occur in the "right" order
  - checked by looking for absence of cycles in precedence graph
- *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.

i.e. there are VS schedules that are not CS, but not vice versa

---

Isolation and Concurrency Control

It is not useful to

- execute a set of tx's T_1 .. T_n to obtain a schedule
- then check whether the schedule produced was serializable

The goal is to devise *concurrency control schemes*

- which arrange the sequence of actions from T_1 .. T_n
- such that we obtain a schedule equivalent to some serial schedule

Serializability tests are used in proving properties of these schemes.

Other Properties of Schedules

Above discussion explicitly assumes that all transactions commit.

What happens if some transaction aborts?

Under serial execution, there is no problem:

- use the log to undo any changes made by T_i
- database is reset to valid state before next T_j starts

With concurrent execution, there may be problems:

- unfortunate interactions with the behaviour of redo/undo log
- inability to recover, even though all schedules are serializable

---

Recoverability

Consider the serializable schedule:

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

(where the final value of `Y` is dependent in 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

The schedule produces an invalid database state, even though serializable.

... 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.

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

Known as *cascading aborts* (or *cascading rollback*).

... Cascading Aborts

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

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

Example: non-strict schedule

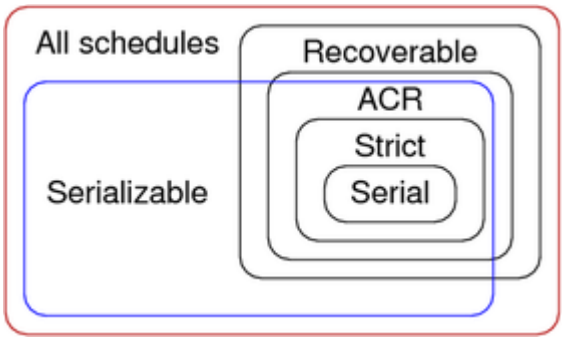
```
T1:  W(X)         A
T2:        W(X)      A
```

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)

Schedule Properties

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 Levels

Previous examples showed:

- if we allow uncontrolled concurrent access to shared data
- the consistency of database may be compromised
- even though individual transactions are consistent

Safest approach ...

- force serial equivalent execution by appropriate locking
- but this costs performance (less concurrency)

Other approaches are weaker ...

- may allow schedules that are not safe
- but provide more opportunity for concurrency

Is a trade-off useful?

... Transaction Isolation Levels

In many real applications, there is either

- no chance for conflict between concurrent transactions
  (e.g. they act on different data or they don't modify the data)
- only a small chance for conflict between transactions
  (in which case it is feasible to abort one and then re-execute it)

This leads to a trade-off between performance and isolation

- determined by programmer on application-by-application basis
- if low concurrency problems, less isolation, better performance
- if significant potential for concurrency problems, more isolation

---

... Transaction Isolation Levels 57/153

SQL provides a mechanism for database programmers to specify how much isolation to apply to transactions

```
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
```

... Transaction Isolation Levels 58/153

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 59/153

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 60/153

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

... Transaction Isolation Levels 61/153

Using PostgreSQL's 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 transaction, either committed or uncommitted)

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

... Transaction Isolation Levels 62/153

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

---

Implementing Concurrency Control

Concurrency Control 64/153

Approaches to concurrency control:

- *Lock-based*

  Synchronise transaction execution via locks on relevant part of DB.

- *Version-based*

  Allow multiple consistent versions of the data to exist.
  Each transaction has exclusive access to one version (the one when tx started).

- *Validation-based* (optimistic concurrency control)

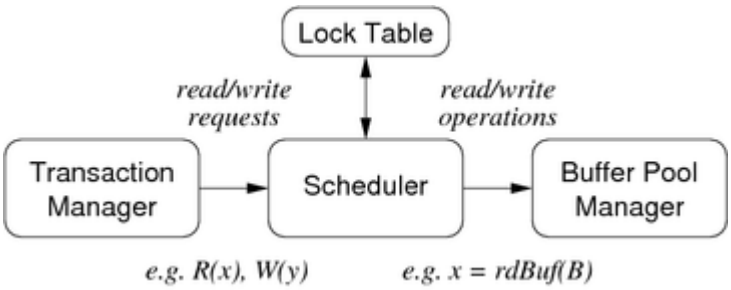
  Execute all transactions; check for validity problems just before commit.

- *Timestamp-based*

Organise transaction execution via timestamps assigned to actions.

Lock–based Concurrency Control

Locks introduce additional mechanisms in DBMS:



The Scheduler

- collects arbitrarily interleaved requests from tx's
- uses locks to delay tx's, if necessary, to avoid unsafe schedules

---

... Lock–based Concurrency Control

Lock table entries contain:

- object being locked
- 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

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 transaction already has write lock on *X*
- a tx attempting to get an write lock on *X* is blocked if another transaction has any kind of lock on *X*

These rules alone do not guarantee serializability.

... Lock–based Concurrency Control

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 shared objects (`X`, `Y`).

Despite this, the schedule is not serializable.

Two-Phase Locking

To guarantee serializability, we require an additional constraint on how locks are applied:

- in every tx, all lock requests precede 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

... Two-Phase Locking

Consider the following two transactions:

T1: L_w(A) R(A) F_1 W(A) L_w(B) U(A) R(B) G_1 W(B) U(B)
T2: L_w(A) R(A) F_2 W(A) L_w(B) U(A) R(B) G_2 W(B) U(B)

They follow 2PL protocol, inducing a schedule like:

T1(a): L_w(A) R(A) F_1 W(A) L_w(B) U(A)
T2(a): L_w(A) ..... R(A) F_2 W(A)

T1(b): R(B) G_1 W(B) U(B)
T2(b): L_w(B) ..... U(A) R(B) G_2 W(B) U(B)

Problems with Locking

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

Deadlock occurs when two transactions are waiting 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 73/153

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 74/153

Simple approach: *timeout*

- set maximum time limit for execution of transaction
- if transaction exceeds limit, abort it
- if caused by deadlock, all its resources are freed

Better approach: *waits-for graph*

- one node per transaction T_i
- a directed edge from T_j to T_k, if
  - T_j is waiting on a lock held by T_k

... Deadlock 75/153

A cycle in the waits-for graph indicates a deadlock.

Could prevent deadlock by

- each time T_i delays trying to get lock (i.e. add new edge to graph)
- check if this would add a cycle to the graph $\Rightarrow$ abort T_i

Could *detect* deadlock by

- periodically check for cycles in the waits-for graph
- choose one T_i from the cycle and abort it

... Deadlock 76/153

Alternative deadlock handling: timestamps.

Each tx is permanently allocated a unique timestamp (e.g. start-time).

When T_j tries to get lock held by T_k

- compare timestamps of two transactions
- use a fixed policy to decide what to do
- two possible policies: *wait-die* or *wound-wait*

Both schemes prevent waits-for cycles by imposing order/priority on tx's.

T_j tries to get lock held by T_k ...

Wait–die scheme:

- if T_j is older than T_k, T_j is allowed to wait
- otherwise, T_j aborts (i.e. is rolled back)

Wound–wait schema:

- if T_j is older than T_k, it "wounds" T_k
  (T_k must roll back and give T_j any locks it needs)
- otherwise, T_j is allowed to wait

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
- timestamp–based are easier to implement than waits–for graph
- waits–for minimises roll backs because of deadlock

Starvation

Starvation occurs when one transaction

- is "trapped" waiting on a lock indefinitely
- while other transactions continue normally

Whether it occurs depends on the lock wait/release strategy.

Multiple locks ⇒ need to decide which to release first.

Solutions:

- implement a fair wait/release strategy (e.g. FIFO)
- use priority deadlock prevention schemes (e.g. wait–die)

Locking Granularity

Locking typically reduces concurrency ⇒ reduces throughput.

Granularity of locking can impact performance:

+ lock a small item ⇒ more of database accessible

+ lock a small item ⇒ quick update ⇒ quick lock release

– lock small items ⇒ more locks ⇒ more lock management

Granularity levels: field, row (tuple), table, whole database

---

... Locking Granularity 81/153

Multiple-granularity locking protocol:

- adds new "intention" locks (L_{IR}, L_{IW})
- before locking a low-level item (e.g. record), acquire intention locks on its ancestors
- unlock from lowest-level to higher-level items

Example: T_1 scans table R and updates some tuples

- gets R+IW lock on R, then gets R lock on each tuple
- occasionally, upgrades to W lock on a tuple

Example: T_2 uses an index to read part of R

- gets IR lock on R, then gets R lock on each tuple

Locking in B-trees 82/153

How to lock a B-tree leaf node?

One possibility:

- lock root, then each node down path
- finally, lock the leaf node

If for searching (select), locks would be read locks.

If for insert/delete, locks would be write locks.

This approach gives poor performance (lock on root is bottleneck).

... Locking in B-trees 83/153

Approach for searching (select) ...

- acquire read lock on each node
- release lock when child has been locked
- repeat down to leaf node (which is only lock still held)

Approach for insert/delete ...

- traverse from root, getting write lock on each node
- at root, check whether any propagation might occur
  (i.e. inserting and node is full, deleteing and node is half full)
- if "safe", release locks on all ancestors
- maintain lock on leaf node and update appropriately

Optimistic Concurrency Control 84/153

Locking is a pessimistic approach to concurrency control:

- limit concurrency to ensure that conflicts don't occur

Costs: lock management, deadlock handling, contention.

In systems where read:write ratio is very high

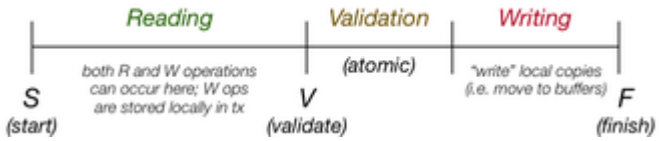
- don't lock (allow arbitrary interleaving of operations)
- check just before commit that no conflicts occurred
- if problems, roll back conflicting transactions

... Optimistic Concurrency Control 85/153

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 86/153

Data structures needed for validation:

- *S = Act set*: txs that are reading data and computing results
- *V = Val set*: txs that have reached validation (not yet committed)
- *F = Fin set*: 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

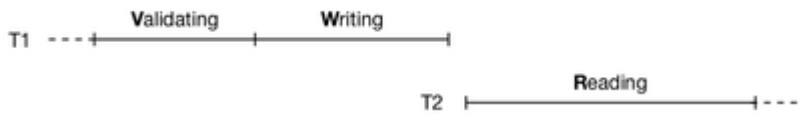
- assume serial tx order based on ordering of $V(T_i)$'s

... Optimistic Concurrency Control 87/153

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



... Optimistic Concurrency Control 88/153

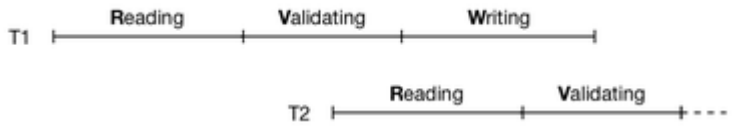
Case 1: reading overlaps validation/writing

- 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

... Optimistic Concurrency Control 89/153

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



... Optimistic Concurrency Control 90/153

Validation check for transaction T

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

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

What this prevents ...

- T reading dirty data (i.e. data later changed by T_i)
- T overwriting changes made by T_i

... Optimistic Concurrency Control 91/153

Problems with optimistic concurrency control:

- if validation fails, "complete" tx's are rolled back
- costs of maintaining the sets of tx's and the R/W–sets

Notes:

- validation test must be atomic ⇒ some locking needed
- T remains in *Fin* until there are no T_i satisfying $A(T_i) < F(T)$

Multi–version Concurrency Control 92/153

Multi-version concurrency control (MVCC) aims to

- retain benefits of locking, but get more concurrency
- provide multiple (consistent) versions of the database

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 93/153

Each transaction is

- associated with a time-stamp (TS)
- declared as a *reader* or a *writer*
  (writers *may* modify data items; readers are guaranteed not to)

Each record in the database is

- tagged with timestamp of tx that wrote it (WTS)
- tagged with timestamp of tx that read it last (RTS)
- chained to older versions of itself
- discarded when it is too old to be "of interest" (vacuum)
  (newer versions exist; all tx's started after the subsequent version)

... Multi-version Concurrency Control 94/153

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 changes a data item

- find newest version V satsifying WTS(V) < TS(T_j)
- if RTS(V) < TS(T_j), create new version of data item
- if no such version available, reject the write

... Multi-version Concurrency Control 95/153

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

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 SQL

Transactions in SQL are specified by

- **`BEGIN`** ... start a transaction
- **`COMMIT`** ... successfully complete a transaction
- **`ROLLBACK`** ... undo changes made by transaction + abort

In PostgreSQL, other actions that cause rollback:

- **raise exception** during execution of a function
- returning null from a **before** trigger

... Concurrency Control in SQL

More fine-grained control of "undo" via savepoints:

- **`SAVEPOINT`** ... marks point in transaction
- **`ROLLBACK TO SAVEPOINT`** ... undo changes, continue transaction

Example:

```
begin;
  insert into numbersTable values (1);
  savepoint my_savepoint;
  insert into numbersTable values (2);
  rollback to savepoint my_savepoint;
  insert into numbersTable values (3);
commit;
```

will insert 1 and 3 into the table, but not 2.

... Concurrency Control in SQL

SQL standard defines four levels of *transaction isolation*.

- *serializable* – strongest isolation, most locking
- *repeatable read*
- *read committed*
- *read uncommitted* – weakest isolation, less locking

The weakest level allows dirty reads, phantom reads, etc.

PostgreSQL implements: repeatable-read = serializable, read-uncommitted = read-committed

---

... Concurrency Control in SQL

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 transaction, either committed or uncommitted)

The transaction containing the failed update will rollback and re-start.

... Concurrency Control in SQL

Explicit control of concurrent access is available, e.g.

Table-level locking: **`LOCK TABLE`**

- various kinds of shared/exclusive locks are available
  - **access share** allows others to read, and some writes
  - **exclusive** allows others to read, but not to write
  - **access exclusive** blocks all other access to table
- SQL commands automatically acquire appropriate locks
  - e.g. **`ALTER TABLE`** acquires an **access exclusive** lock

Row-level locking: **`SELECT FOR UPDATE`**, **`DELETE`**

- allows others to read, but blocks write on selected rows

All locks are released at end of transaction (no explicit unlock)

Concurrency Control in PostgreSQL

PostgreSQL uses two styles of concurrency control:

- multi-version concurrency control (MVCC)
  (used in the implementation of SQL DML statements (e.g. `select`))
- two-phase locking (2PL)
  (used in the implementation of 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

Implementing MVCC in PostgreSQL requires:

- a log file to maintain current status of each T_j
- 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_j:
    - a transaction ID (timestamp)
    - SnapshotData: list of active tx's when T_j started

... Concurrency Control in PostgreSQL 104/153

Rules for a tuple to be visible to T_j:

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

... Concurrency Control in PostgreSQL 105/153

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; 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

Implementing Atomicity/Durability

Atomicity/Durability 107/153

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

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

Consider following scenario:

Desired behaviour after system restart:

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

... Durability

Durabilty begins with a *stable disk storage subsystem*.

Operations:

- `putBlock(Buffer *b)` ... writes data from buffer to disk
- `getBlock(Buffer *b)` ... reads data from disk into buffer

Each call to `putBlock` must ensure that

- data is transferred from buffer to disk verbatim (unchanged)

Subsequent `getBlock` on same disk sector must ensure that

- same data that was last written is transferred back to buffer

---

... Durability

Implementation of transaction operations *R(V)* and *W(V)*

```
Value R(Object V) {
    B = getBuf(blockContaining(V))
    return value of V from B
}
void W(Object V, value k) {
    B = getBuf(blockContaining(V))
    set value of V in B
}
```

Note:

- *W* does not actually do output; happens when buffer replaced
- if tx terminates before buffer replaced, need to do `putBuf()`

... Durability

`getBuf()` and `putBuf()` interface buffer pool with disk

```
Buffer getBuf(BlockAddr A) {
    if (!inBufferPool(A)) {
        B = availableBuffer(A);
        getBlock(B);
    }
    return bufferContaining(A);
}
void putBuf(BlockAddr A) {
    B = bufferContaining(A);
    putBlock(B);
}
```

Stable Store

One simple strategy using redundancy: *stable store*.

Protects against all failures in a single disk sector.

Each logical disk page *X* is stored twice.

(Obvious disadvantage: disk capacity is halved)

X is stored in sector *S* on disk *L* and sector *T* on disk *R*

Assume that a sound parity–check is available

(i.e. can always recognise whether data has transferred mem↔disk correctly)

... Stable Store

Low level sector i/o functions:

```
int writeSector(char *b, Disk d, Sector s) {
   int nTries = 0;
   do {
      nTries++;  write(d,s,b);
   } while (bad(parity) && nTries < MaxTries)
   return nTries;
}
int readSector(char *b, Disk d, Sector s) {
   int nTries = 0;
   do {
      nTries++;  read(d,s,b);
   } while (bad(parity) && nTries < MaxTries)
   return nTries;
}
```

... Stable Store

Writing data to disk with stable store:

```
int writeBlock(Buffer *b, Disk d, Sector s) {
   int sec;
   for (;;) {
      sec = (s > 0) ? s : getUnusedSector(d);
      n = writeSector(b->data, d, sec);
      if (n == maxTries)
         mark s as unusable
      else
         return sec;
   }
}
```

- call `writeBlock` twice for each buffer *b*, on disks *L* and *R*
- remember association between *b* and *(L,S)* and *(R,T)*

... Stable Store 116/153

Reading data from disk with stable store:

```
int readBlock(Buffer *b) {
   int n = readSector(b->data, b->diskL, b->sectorL);
   if (n == maxTries) {
      n = readSector(b->data, b->diskR, b->sectorR);
      if (n == maxTries) return -1; // read failure
   }
   return 0; // successful read
}
```

- if read from disk *L* succeeds, ignore disk *R*
  (if sector *(R,T)* has failed, we discover failure on subsequent write)
- if read from disk *L* fails, get correct copy from disk *R*
  (and, at the same time, can mark sector *(L,S)* as bad)
- the chances of both *(L,S)* and *(R,T)* failing is extremely small

---

... Stable Store 117/153

Consider scenario where power fails *during* write operation:

- aim is to write new contents of buffer *X* to stable store
- will be done via `writeBlock(X,L,S); writeBlock(X,R,T);`
- on power–fail, intended new *X* is lost from memory
- partial write to sector will produce corrupted data
- power–fail happens at a point in time, so only one write fails

Wish to restore the system to a state where:

- sectors X_L and X_R are back "in sync"
- can recognize whether we have old/new version on disk

... Stable Store 118/153

How stable storage handles failure during writing:

- failure occurs while writing X_L ...
  - sector X_L will subsequently be seen as bad
  - valid, but old, value of *X* is available in X_R
  - can repair X_L by copying value from X_R
- failure occurs while writing X_R ...
  - sector X_R will subsequently be seen as bad

- new value of X is available in X_L
- can repair X_R by copying value from X_L

RAID

RAID gives techniques to achieve

- good read/write performance
- recovery from multiple simultaneous disk failures

Requires:

- multiple disks (Redunant Array of Inexpensive Disks)
- arrangement of data across multiple disks
- use of error–correcting codes (ECCs)

(See texts for further discussion on RAID)

Stable Storage Subsystem

We can prevent/minimise loss/corruption of data due to:

- mem/disk transfer corruption: parity checking
- sector failure: mark "bad" blocks, stable storage
- disk failure: RAID (levels 4,5,6)
- destruction of computer system:
  – complete DB backups, stored away from computer system
  – on–line, distributed copies of database (consistency?)

If all of these implemented, assume stable storage subsystem.

Dealing with Transactions

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

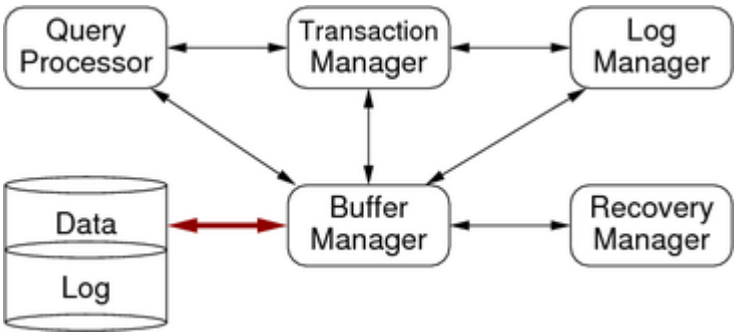
- failure of DBMS processes or operating system
- failure of transactions (`ABORT`)

Standard technique for managing these:

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

Architecture for Atomicity/Durability

How does a DBMS provide for atomicity/durability?



Execution of Transactions

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.

... Execution of Transactions

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).

... Execution of Transactions

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

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

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

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

... Transactions and Buffer Pool

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

Above scenario may be a problem, even if we are forcing

- e.g. system crashes immediately after requesting a `WRITE()`

Logging

Three "styles" of logging

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

All approaches require:

- a sequential file of log records
- records describing changes are written first
- actual changes to data are written later

Undo Logging

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:

- 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. the 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
                     { 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
```

... Undo Logging 135/153

Recall example transaction and consider effects of system crash at the following points:

Before (9) ... disk "restored" (unchanged); `<ABORT T>` written

(9)–(11) ... disk restored to original state; `<ABORT T>` written

After (12) ... A and B left unchanged; `T` treated as committed

"Disk restored" means

```
WRITE(B,5); OUTPUT(B); WRITE(A,8); OUTPUT(A);
```

Checkpointing 136/153

Previous view of recovery implied 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 137/153

Problem: many concurrent/overlapping transactions.

How to know that all have finished?

Simplistic approach

1. stop accepting new transactions (block them)
2. wait until all active tx's either commit or abort
   (and have written a `<COMMIT T>` or `<ABORT T>` on the log)
3. flush the log to disk
4. write new log record `<CHKPT>`, and flush log again
5. resume accepting new transactions

Known as *quiescent checkpointing*.

... Checkpointing

Obvious problem with quiescent checkpointing

- DBMS effectively shut down until all existing tx's finish

Better strategy: *nonquiescent checkpointing*

1. 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

... Checkpointing

Recovery: scan backwards through log file processing as before.

Determining where to stop depends on ...

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 done
- for uncommitted tx's, need to continue backward scan
  (could simplify this task by chaining together log records for each tx)

Redo Logging

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

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. the commit log record

4. the changed data elements themselves

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

... Redo 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,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.

... Redo Logging

Simplified view of recovery using REDO logging:

```
set committedTrans // e.g. from tx table
for each log record from oldest to most recent {
    switch (log record) {
    <COMMIT T> : ignore // know these already
    <ABORT T>  : add T to abortedTrans
    <START T>  : add T to startedTrans
    <T,X,v'>   : if (T in committedTrans)
                       { WRITE(X,v'); OUTPUT(X) }
                 else
                     // nothing to do, no change on disk
}   }
for each T in startedTrans {
    if (T in committedTrans) ignore
    else if (T in abortedTrans) ignore
    else write <ABORT T> to log
}
flush log
```

... Redo Logging

Data required for REDO logging checkpoints:

- must know which transactions have committed
  (hold a list of completed tx's in tx table between checkpoints)
- must know which buffer pool pages are dirty
  (buffer pool would normally record this information anyway)
- must know which tx's modified these pages
  (one buffer pool page may have been written to by several tx's)

... Redo Logging

Checkpoints in REDO logging use (as before):

- `<CHKPT (T1,..,Tk)>` to record active tx's at start of checkpoint
- `<ENDCHKPT>` to record end of checkpoint period

Steps in REDO log checkpointing

1. write `<CHKPT (T1,..,Tk)>` to log and flush log
2. output to disk all changed objects in buffer pool
   which have not yet been written to disk
   whose tx's had committed before `<CHKPT...>`
3. write `<ENDCHKPT>` to log and flush log

Note that other tx's may continue between steps 2 and 3.

... Redo Logging

Recovery with checkpointed REDO log.

As for UNDO logging, two cases ...

Last checkpoint record before crash is `<ENDCHKPT>`

- every tx that committed before prev `<CHKPT...>` is fine
- need to restore values for all `T1,..,Tk` (long backward search?)
- similarly for all Ti started after `<CHKPT...>`

Last checkpoint record before crash is `<CHKPT (T1,..,Tk)>`

- cannot be sure that committed tx's before here are done
- need to search back to most recent prior `<ENDCHKPT>`
- then proceed as for first case

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.

... Undo/Redo Logging

Requirement for undo/redo logging: *write–ahead*.

Data must be written to disk as follows:

1. start transaction log record
2. update log records indicating changes
3. the changed data elements themselves

Do not specify when the <COMMIT T> record is written.

... Undo/Redo Logging

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.

... Undo/Redo Logging

Recovery using undo/redo logging:

1. redo all committed tx's from earliest to latest
2. undo all incomplete tx's from latest to earliest

Consider effects of system crash on example

Before (10) ... treat as incomplete; undo all changes

After (10) ... treat as complete; redo all changes

... Undo/Redo Logging

Steps in UNDO/REDO log checkpointing

1. write <CHKPT (T1,..,Tk)> to log and flush log
2. output to disk *all* dirty memory buffers
3. write <ENDCHKPT> to log and flush log

Note that other tx's may continue between steps 2 and 3.

A consequence of the above:

- tx's must *not* place changed data in buffers
  until they are certain that they will COMMIT

(If we allowed this, a buffer may contain changes from both committed and aborted tx's)

... Undo/Redo Logging

The above description simplifies details of undo/redo logging.

Aries is a complete algorithm for undo/redo logging.

Differences to what we have described:

- log records contain a sequence numnber (LSN)
- LSNs used in tx and buffer managers, and stored in data pages
- additional log record to mark `<END>` (of commit or abort)
- `<CHKPT>` contains only a timestamp
- `<ENDCHKPT..>` contains tx and dirty page info

(For more details consult any text or COMP9315 05s1 lecture notes)

Recovery in PostgreSQL

PostgreSQL uses write–ahead undo/redo style logging.

However, it also uses multi–version concurrency control

- tags each record with a tx and update timestamp
- which simplifies 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; old versions still available

Transaction/logging code is distributed throughout backend source.

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

Produced: 7 Aug 2019