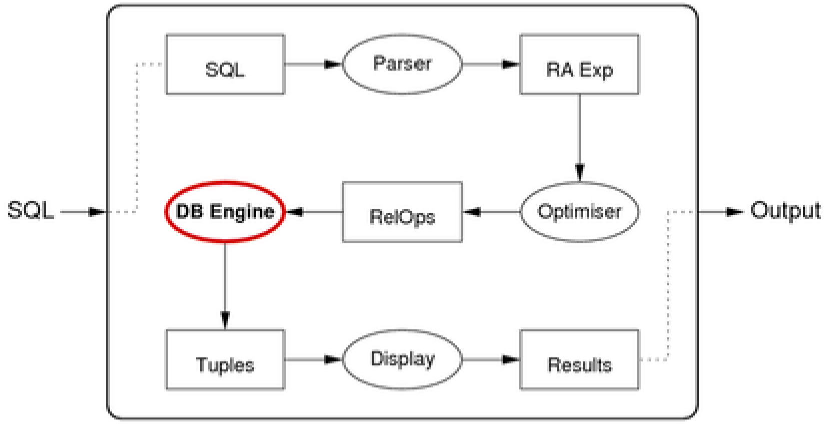
Week 09

Query Execution

Query Execution

Query execution: applies evaluation plan → result tuples



... Query Execution

Example of query translation:

```
select s.name, s.id, e.course, e.mark
from   Student s, Enrolment e
where  e.student = s.id and e.semester = '05s2';
```

maps to

$$\pi_{name,id,course,mark}(Stu \Join_{e.student=s.id} (\sigma_{semester=05s2}Enr))$$

maps to

```
Temp1  = BtreeSelect[semester=05s2](Enr)
Temp2  = HashJoin[e.student=s.id](Stu,Temp1)
Result = Project[name,id,course,mark](Temp2)
```

... Query Execution

A query execution plan:

- consists of a *collection of RelOps*
- executing together to produce a set of result tuples

Results may be passed from one operator to the next:

- *materialization* ... writing results to disk and reading them back
- *pipelining* ... generating and passing via memory buffers

Materialization

Steps in *materialization* between two operators

- first operator reads input(s) and writes results to disk
- next operator treats tuples on disk as its input
- in essence, the `Temp` tables are produced as real tables

Advantage:

- intermediate results can be placed in a file structure
  (which can be chosen to speed up execution of subsequent operators)

Disadvantage:

- requires disk space/writes for intermediate results
- requires disk access to read intermediate results

Pipelining

How *pipelining* is organised between two operators:

- operators execute "concurrently" as producer/consumer pairs
- structured as interacting iterators (open; while(next); close)

Advantage:

- no requirement for disk access (results passed via memory buffers)

Disadvantage:

- higher-level operators access inputs via linear scan, or
- requires sufficient memory buffers to hold all outputs

Iterators (reminder)

Iterators provide a "stream" of results:

- **`iter = startScan(`*params*`)`**
    - set up data structures for iterator (create state, open files, ...)
    - *params* are specific to operator (e.g. reln, condition, #buffers, ...)
- **`tuple = nextTuple(iter)`**
    - get the next tuple in the iteration; return null if no more
- **`endScan(iter)`**
    - clean up data structures for iterator

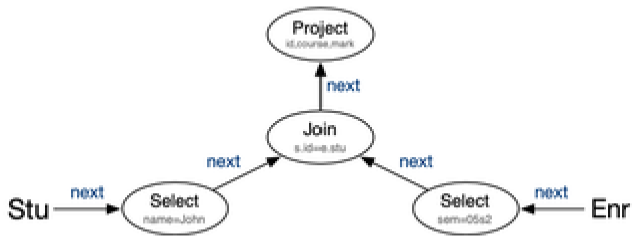
Other possible operations: reset to specific point, restart, ...

Pipelining Example

Consider the query:

```
select s.id, e.course, e.mark
from   Student s, Enrolment e
where  e.student = s.id and
       e.semester = '05s2' and s.name = 'John';
```

Evaluated via communication between RA tree nodes:



Note: likely that projection is combined with join in PostgreSQL

Disk Accesses

9/136

Pipelining cannot avoid all disk accesses.

Some operations use multiple passes (e.g. merge-sort, hash-join).

- data is written by one pass, read by subsequent passes

Thus ...

- *within* an operation, disk reads/writes are possible
- *between* operations, no disk reads/writes are needed

PostgreSQL Query Execution

PostgreSQL Query Execution

11/136

Defs: **`src/include/executor`** and **`src/include/nodes`**

Code: **`src/backend/executor`**

PostgreSQL uses pipelining (as much as possible) ...

- query plan is a tree of **`Plan`** nodes
- each type of node implements one kind of RA operation
  (node implements specific access method via iterator interface)
- node types e.g. **`Scan`**, **`Group`**, **`Indexscan`**, **`Sort`**, **`HashJoin`**
- execution is managed via a tree of **`PlanState`** nodes
  (mirrors the structure of the tree of Plan nodes; holds execution state)

PostgreSQL Executor

12/136

Modules in **`src/backend/executor`** fall into two groups:

`execXXX` (e.g. execMain, execProcnode, execScan)

- implement generic control of plan evaluation (execution)
- provide overall plan execution and dispatch to node iterators

`nodeXXX` (e.g. nodeSeqscan, nodeNestloop, nodeGroup)

- implement iterators for specific types of RA operators
- typically contains **`ExecInitXXX`**, **`ExecXXX`**, **`ExecEndXXX`**

Example PostgreSQL Execution

Consider the query:

```
-- get manager's age and # employees in Shoe department
select e.age, d.nemps
from   Departments d, Employees e
where  e.name = d.manager and d.name ='Shoe'
```

and its execution plan tree

... Example PostgreSQL Execution

Initially `InitPlan()` invokes `ExecInitNode()` on plan tree root.

`ExecInitNode()` sees a `NestedLoop` node ...
  so dispatches to `ExecInitNestLoop()` to set up iterator
  then invokes `ExecInitNode()` on left and right sub-plans
    in left subPlan, `ExecInitNode()` sees an `IndexScan` node
     so dispatches to `ExecInitIndexScan()` to set up iterator
    in right sub-plan, `ExecInitNode()` sees a `SeqScan` node
     so dispatches to `ExecInitSeqScan()` to set up iterator

Result: a plan state tree with same structure as plan tree.

... Example PostgreSQL Execution

Then `ExecutePlan()` repeatedly invokes `ExecProcNode()`.

`ExecProcNode()` sees a `NestedLoop` node ...
  so dispatches to `ExecNestedLoop()` to get next tuple
  which invokes `ExecProcNode()` on its sub-plans
    in left sub-plan, `ExecProcNode()` sees an `IndexScan` node
      so dispatches to `ExecIndexScan()` to get next tuple
      if no more tuples, return END
      for this tuple, invoke `ExecProcNode()` on right sub-plan
        `ExecProcNode()` sees a `SeqScan` node
          so dispatches to `ExecSeqScan()` to get next tuple
          check for match and return joined tuples if found
          continue scan until end
        reset right sub-plan iterator

Result: stream of result tuples returned via `ExecutePlan()`

Query Performance

Performance Tuning

How to make a database-backed system perform "better"?

Improving performance may involve any/all of:

- making applications using the DB run faster
- lowering response time of queries/transactions
- improving overall transaction throughput

Remembering that, to some extent ...

- the query optimiser removes choices from DB developers
- by making its own decision on the optimal execution plan

... Performance Tuning

Tuning requires us to consider the following:

- which queries and transactions will be used?
  (e.g. check balance for payment, display recent transaction history)
- how frequently does each query/transaction occur?
  (e.g. 80% withdrawals; 1% deposits; 19% balance check)
- are there time constraints on queries/transactions?
  (e.g. EFTPOS payments must be approved within 7 seconds)
- are there uniqueness constraints on any attributes?
  (define indexes on attributes to speed up insertion uniqueness check)
- how frequently do updates occur?
  (indexes slow down updates, because must update table *and* index)

... Performance Tuning

Performance can be considered at two times:

- *during* schema design
  - typically towards the end of schema design process
  - requires schema transformations such as *denormalisation*
- *outside* schema design
  - typically after application has been deployed/used
  - requires adding/modifying data structures such as *indexes*

Difficult to predict what query optimiser will do, so ...

- implement queries using methods which *should* be efficient
- observe execution behaviour and modify query accordingly

PostgreSQL Query Tuning

PostgreSQL provides the **`explain`** statement to

- give a representation of the query execution plan
- with information that may help to tune query performance

Usage:

`EXPLAIN [ANALYZE]` *`Query`*

Without `ANALYZE`, `EXPLAIN` shows plan with estimated costs.

With `ANALYZE`, `EXPLAIN` executes query and prints real costs.

Note that runtimes may show considerable variation due to buffering.

EXPLAIN Examples

Database

```
people(id, family, given, title, name, ..., birthday)
courses(id, subject, semester, homepage)
course_enrolments(student, course, mark, grade, ...)
subjects(id, code, name, longname, uoc, offeredby, ...)
...
```

where

```
       table_name           | n_records
---------------------------+-----------
 people                     |    150963
 courses                    |     34955
 course_enrolments          |   1812317
 subjects                   |     33377
...
```

... EXPLAIN Examples

Example: Select on non-indexed attribute

```
uni=# explain
uni=# select * from Students where stype='local';
                     QUERY PLAN
----------------------------------------------------
 Seq Scan on students
             (cost=0.00..562.01 rows=23544 width=9)
   Filter: ((stype)::text = 'local'::text)
```

where

- `Seq Scan` = operation (plan node)
- `cost=`*StartUpCost*`..`*TotalCost*
- `rows=`*NumberOfResultTuples*
- `width=`*SizeOfTuple* (# bytes)

... EXPLAIN Examples

More notes on `explain` output:

- each major entry corresponds to a plan node
  - e.g. `Seq Scan`, `Index Scan`, `Hash Join`, `Merge Join`, ...
- some nodes include additional qualifying information
  - e.g. `Filter`, `Index Cond`, `Hash Cond`, `Buckets`, ...
- `cost` values in `explain` are estimates (notional units)
- `explain analyze` also includes actual time costs (ms)
- costs of parent nodes include costs of all children
- estimates of #results based on sample of data

... EXPLAIN Examples

Example: Select on non-indexed attribute with actual costs

```
uni=# explain analyze
uni=# select * from Students where stype='local';
                       QUERY PLAN
```

```
----------------------------------------------------------
 Seq Scan on students
             (cost=0.00..562.01 rows=23544 width=9)
             (actual time=0.052..5.792 rows=23551 loops=1)
   Filter: ((stype)::text = 'local'::text)
   Rows Removed by Filter: 7810
 Planning time: 0.075 ms
 Execution time: 6.978 ms
```

... EXPLAIN Examples

Example: Select on indexed, unique attribute

```
uni=# explain analyze
uni-# select * from Students where id=100250;
                       QUERY PLAN
-------------------------------------------------------
 Index Scan using student_pkey on student
            (cost=0.00..8.27 rows=1 width=9)
            (actual time=0.049..0.049 rows=0 loops=1)
   Index Cond: (id = 100250)
 Total runtime: 0.1 ms
```

... EXPLAIN Examples

Example: Select on indexed, unique attribute

```
uni=# explain analyze
uni-# select * from Students where id=1216988;
                       QUERY PLAN
-------------------------------------------------------
 Index Scan using students_pkey on students
                  (cost=0.29..8.30 rows=1 width=9)
                  (actual time=0.011..0.012 rows=1 loops=1)
   Index Cond: (id = 1216988)
 Planning time: 0.066 ms
 Execution time: 0.026 ms
```

... EXPLAIN Examples

Example: Join on a primary key (indexed) attribute (2016)

```
uni=# explain analyze
uni-# select s.id,p.name
uni-# from Students s, People p where s.id=p.id;
                      QUERY PLAN
----------------------------------------------------------
Hash Join (cost=988.58..3112.76 rows=31048 width=19)
          (actual time=11.504..39.478 rows=31048 loops=1)
  Hash Cond: (p.id = s.id)
  -> Seq Scan on people p
         (cost=0.00..989.97 rows=36497 width=19)
         (actual time=0.016..8.312 rows=36497 loops=1)
  -> Hash (cost=478.48..478.48 rows=31048 width=4)
          (actual time=10.532..10.532 rows=31048 loops=1)
          Buckets: 4096  Batches: 2  Memory Usage: 548kB
      -> Seq Scan on students s
              (cost=0.00..478.48 rows=31048 width=4)
              (actual time=0.005..4.630 rows=31048 loops=1)
Total runtime: 41.0 ms
```

... EXPLAIN Examples

Example: Join on a primary key (indexed) attribute (2018)

```
uni=# explain analyze
uni-# select s.id,p.name
uni-# from Students s, People p where s.id=p.id;
                      QUERY PLAN
----------------------------------------------------------
Merge Join  (cost=0.58..2829.25 rows=31361 width=18)
            (actual time=0.044..25.883 rows=31361 loops=1)
  Merge Cond: (s.id = p.id)
  ->  Index Only Scan using students_pkey on students s
            (cost=0.29..995.70 rows=31361 width=4)
            (actual time=0.033..6.195 rows=31361 loops=1)
        Heap Fetches: 31361
  ->  Index Scan using people_pkey on people p
            (cost=0.29..2434.49 rows=55767 width=18)
            (actual time=0.006..6.662 rows=31361 loops=1)
Planning time: 0.259 ms
Execution time: 27.327 ms
```

... EXPLAIN Examples

Example: Join on a non-indexed attribute (2016)

```
uni=# explain analyze
uni=# select s1.code, s2.code
uni-# from Subjects s1, Subjects s2
uni=# where s1.offeredBy=s2.offeredBy;
                        QUERY PLAN
---------------------------------------------------------------
Merge Join (cost=4449.13..121322.06 rows=7785262 width=18)
           (actual time=29.787..2377.707 rows=8039979 loops=1)
 Merge Cond: (s1.offeredby = s2.offeredby)
 ->  Sort (cost=2224.57..2271.56 rows=18799 width=13)
          (actual time=14.251..18.703 rows=18570 loops=1)
     Sort Key: s1.offeredby
     Sort Method: external merge  Disk: 472kB
     ->  Seq Scan on subjects s1
             (cost=0.00..889.99 rows=18799 width=13)
             (actual time=0.005..4.542 rows=18799 loops=1)
 ->  Sort (cost=2224.57..2271.56 rows=18799 width=13)
          (actual time=15.532..1100.396 rows=8039980 loops=1)
     Sort Key: s2.offeredby
     Sort Method: external sort  Disk: 552kB
     ->  Seq Scan on subjects s2
             (cost=0.00..889.99 rows=18799 width=13)
             (actual time=0.002..3.579 rows=18799 loops=1)
Total runtime: 2767.1 ms
```

... EXPLAIN Examples

Example: Join on a non-indexed attribute (2018)

```
uni=# explain analyze
uni=# select s1.code, s2.code
uni-# from Subjects s1, Subjects s2
uni-# where s1.offeredBy = s2.offeredBy;
                        QUERY PLAN
---------------------------------------------------------------
Hash Join  (cost=1286.03..108351.87 rows=7113299 width=18)
           (actual time=8.966..903.441 rows=7328594 loops=1)
  Hash Cond: (s1.offeredby = s2.offeredby)
  ->  Seq Scan on subjects s1
          (cost=0.00..1063.79 rows=17779 width=13)
          (actual time=0.013..2.861 rows=17779 loops=1)
  ->  Hash  (cost=1063.79..1063.79 rows=17779 width=13)
            (actual time=8.667..8.667 rows=17720 loops=1)
        Buckets: 32768  Batches: 1  Memory Usage: 1087kB
        ->  Seq Scan on subjects s2
```

```
                (cost=0.00..1063.79 rows=17779 width=13)
                (actual time=0.009..4.677 rows=17779 loops=1)
Planning time: 0.255 ms
Execution time: 1191.023 ms
```

... EXPLAIN Examples

Example: Join on a non-indexed attribute (2018)

```
uni=# explain analyze
uni=# select s1.code, s2.code
uni-# from Subjects s1, Subjects s2
uni-# where s1.offeredBy = s2.offeredBy and s1.code < s2.code;
                        QUERY PLAN
---------------------------------------------------------------
Hash Join  (cost=1286.03..126135.12 rows=2371100 width=18)
           (actual time=7.356..6806.042 rows=3655437 loops=1)
  Hash Cond: (s1.offeredby = s2.offeredby)
  Join Filter: (s1.code < s2.code)
  Rows Removed by Join Filter: 3673157
  ->  Seq Scan on subjects s1
          (cost=0.00..1063.79 rows=17779 width=13)
          (actual time=0.009..4.602 rows=17779 loops=1)
  ->  Hash  (cost=1063.79..1063.79 rows=17779 width=13)
            (actual time=7.301..7.301 rows=17720 loops=1)
        Buckets: 32768  Batches: 1  Memory Usage: 1087kB
        ->  Seq Scan on subjects s2
                (cost=0.00..1063.79 rows=17779 width=13)
                (actual time=0.005..4.452 rows=17779 loops=1)
Planning time: 0.159 ms
Execution time: 6949.167 ms
```

Exercise 1: EXPLAIN examples

Using the database described earlier ...

```
Course_enrolments(student, course, mark, grade, ...)
Courses(id, subject, semester, homepage)
People(id, family, given, title, name, ..., birthday)
Program_enrolments(id, student, semester, program, wam, ...)
Students(id, stype)
Subjects(id, code, name, longname, uoc, offeredby, ...)

create view EnrolmentCounts as
 select s.code, c.semester, count(e.student) as nstudes
   from Courses c join Subjects s on c.subject=s.id
        join Course_enrolments e on e.course = c.id
  group by s.code, c.semester;
```

predict how each of the following queries will be executed ...

Check your prediction using the `EXPLAIN ANALYZE` command.

1. select max(birthday) from People
2. select max(id) from People
3. select family from People order by family
4. select distinct p.id, pname
   from People s, CourseEnrolments e
   where s.id=e.student and e.grade='FL'
5. select * from EnrolmentCounts where code='COMP9315';

Examine the effect of adding `ORDER BY` and `DISTINCT`.

Add indexes to improve the speed of slow queries.

Transaction Processing

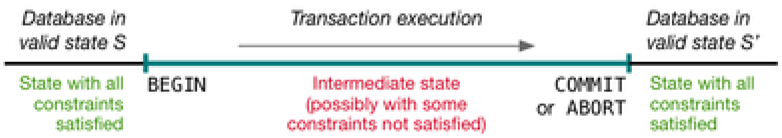
Transaction Processing

35/136

A *transaction* (tx) is ...

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

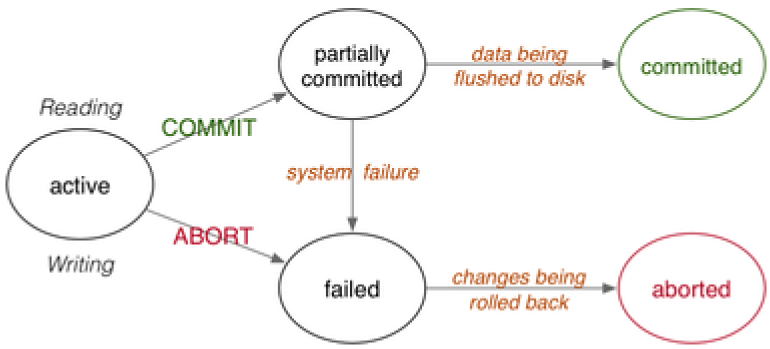
A transaction effects a state change on the DB



... Transaction Processing

36/136

Transaction states:



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

... Transaction Processing

37/136

Concurrent transactions are

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

To ensure problem-free concurrent transactions:

- Atomic ... whole effect of tx, or nothing
- Consistent ... individual tx's are "correct" (wrt application)
- Isolated ... each tx behaves as if no concurrency
- Durable ... effects of committed tx's persist

... Transaction Processing

38/136

Transaction processing:

- the study of techniques for realising ACID properties

Consistency is the property:

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

Ensuring this must be left to application programmers.

Our discussion focusses on: Atomicity, Durability, Isolation

... Transaction Processing

Atomicity is handled by the *commit* and *abort* mechanisms

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

Durability is handled by implementing *stable storage*, via

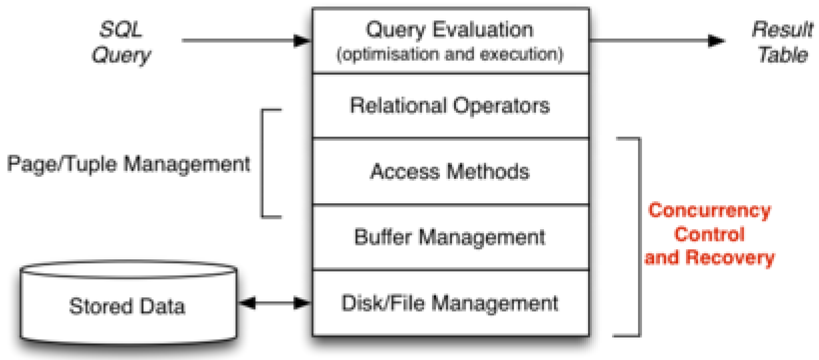
- redundancy, to deal with hardware failures
- logging/checkpoint mechanisms, to recover state

Isolation is handled by *concurrency control* mechanisms

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

... Transaction Processing

Where transaction processing fits in the DBMS:



Transaction Terminology

To describe transaction effects, we consider:

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

Relationship between the above operations and SQL:

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

- **`INSERT`** produces `WRITE` operations

... Transaction Terminology

More on transactions and SQL

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

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

... Transaction Terminology

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

- occur in the context of some transaction T
- involve manipulation of data items $X, Y, \ldots$ (`READ` and `WRITE`)

The operations are typically denoted as:

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

Schedules

A *schedule* gives the sequence of operations from $\geqslant 1$ tx

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

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

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

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

- operations from individual T_i's are interleaved

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

... Schedules

Serial schedules guarantee database consistency

- each T_i commits before T_{i+1}
- prior to T_i database is consistent

- after T_i database is consistent (assuming T_i is correct)
- before T_{i+1} database is consistent ...

Concurrent schedules interleave tx operations arbitrarily

- and may produce a database that is not consistent
- even after all of the tx's have committed successfully

Transaction Anomalies

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

The set of phenomena can be characterised broadly under:

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

Exercise 2: Update Anomaly

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

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

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

Exercise 3: How many Schedules?

In the previous exercise, we looked at several schedules

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

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

Schedule Properties

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

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

Primary goal of *isolation* mechanisms (see later) is

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

Serializability is one property of a schedule, focusing on isolation

Other properties of schedules focus on recovering from failures

---

Transaction Failure

So far, have implicitly assumed that all transactions commit.

Additional problems can arise when transactions abort.

Consider the following schedule where transaction T1 fails:

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

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

Consider three places where the rollback might occur:

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

... Transaction Failure

Abort / rollback scenarios:

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

Case [1] is ok

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

Case [2] is problematic

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

Case [3] is also problematic

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

Recoverability

Consider the serializable schedule:

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

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

Notes:

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

Produces an invalid database state, even though serializable.

... Recoverability

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.

Exercise 4: Recoverability/Serializability

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

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

Consider the two transactions:

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

Give examples of schedules on T1 and T2 that are

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

Cascading Aborts

Recall the earlier non-recoverable schedule:

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

To make it recoverable requires:

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

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

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

... Cascading Aborts

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

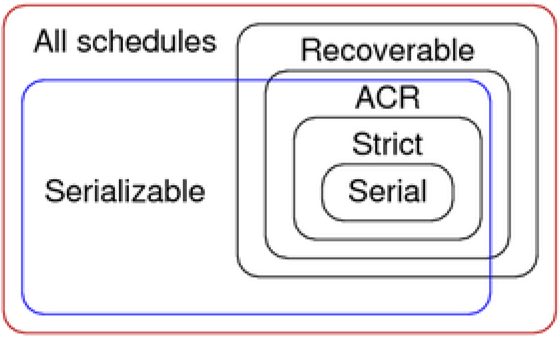
```
T1:  W(X)        A
T2:       W(X)      A
```

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)

Classes of Schedules

Relationship between various classes of schedules:



Schedules ought to be serializable and strict.

But more serializable/strict ⇒ less concurrency.

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

Transaction Isolation

Transaction Isolation

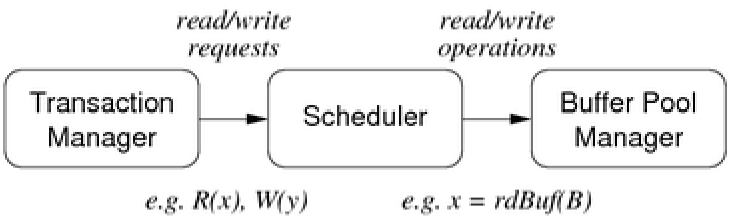
62/136

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

63/136

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

64/136

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 5: Serializability Checking

Is the following schedule view/conflict serializable?

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

Is the following schedule view/conflict serializable?

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

Transaction Isolation Levels

SQL programmers' concurrency control mechanism ...

```
set transaction
    read only  -- so weaker isolation may be ok
    read write -- suggests stronger isolation needed
isolation level
    -- weakest isolation, maximum concurrency
    read uncommitted
    read committed
    repeatable read
    serializable
    -- strongest isolation, minimum concurrency
```

Applies to current tx only; affects how scheduler treats this tx.

... Transaction Isolation Levels

Implication of transaction isolation levels:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible

Serializable	Not Possible	Not Possible	Not Possible

For transaction isolation, PostgreSQL

- provides syntax for all four levels
- treats *read uncommitted* as *read committed*
- *repeatable read* behaves *like serializable*
- default level is *read committed*

Note: cannot implement *read uncommitted* because of MVCC

For more details, see PostgreSQL Documentation section 13.2

- extensive discussion of semantics of `UPDATE`, `INSERT`, `DELETE`

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

Transactions fail if the system detects violation of isolation level.

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

Aims of *concurrency control* schemes

- each transaction behaves as if it's the only running tx
- as much as possible, avoid transaction anomalies
- provide as much concurrency as possible (performance)

As the name suggests, these schemes aim to *control* concurrency

- ensure that op's from concurrent tx's occur in a "safe" order
- if "unsafe" detected, need to rollback ⩾ 1 transactions

... Concurrency Control

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

Synchronise access to shared data items via following rules:

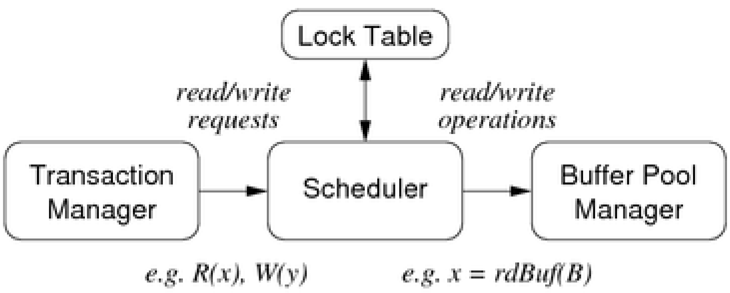
- before reading *X*, get *read lock* on *X* (shared)
- before writing *X*, get *write lock* on *X* (exclusive)
- a tx attempting to get a read lock on *X* is blocked
  if another tx already has write lock on *X*
- a tx attempting to get an write lock on *X* is blocked
  if another tx has any kind of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

Blocking causes transactions to wait ⇒ reduces concurrency

But also prevents some transaction aomalies

... Lock-based Concurrency Control

Locks introduce additional mechanisms in DBMS:



The Lock Manager

- manages the locks requested by the scheduler

... Lock-based Concurrency Control

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

Consider the following schedule, using locks:

T1(a): L_r(Y)   R(Y)   continued
T2(a): L_r(X)   R(X) U(X)   continued

T1(b):   U(Y) L_w(X) W(X) U(X)
T2(b): L_w(Y)....W(Y) U(Y)

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

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

Despite this, the schedule is not serializable. (Ex: prove this)

Two-Phase Locking

To guarantee serializability, we require an additional constraint:

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

Each transaction is then structured as:

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

Clearly reduces potential concurrency

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: 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

Handling deadlock involves forcing a transaction to "back off"

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

... Deadlock

Methods for managing deadlock

- *timeout* : set max time limit for each tx
- *waits-for graph* : records T_j waiting on lock held by T_k
  - *prevent* deadlock by checking for new cycle ⇒ abort T_i
  - *detect* deadlock by periodic check for cycles ⇒ abort T_i
- *timestamps* : use tx start times as basis for priority
  - scenario: T_j tries to get lock held by T_k ...
  - *wait-die*: if $T_j < T_k$, then T_j waits, else T_j rolls back
  - *wound-wait*: if $T_j < T_k$, then T_k rolls back, else T_j waits

---

... Deadlock

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 6: Deadlock Handling

Consider the following schedule on four transactions:

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

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

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

Show how any deadlocks might be resolved via this graph.

Optimistic Concurrency Control

Locking is a pessimistic approach to concurrency control:

- limit concurrency to ensure that conflicts don't occur

Costs: lock management, deadlock handling, contention.

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

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

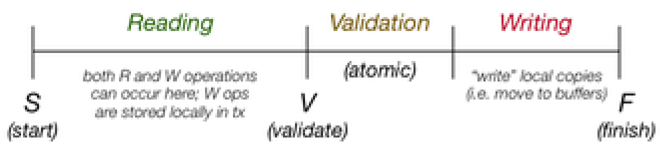
Optimistic concurrency control (OCC) is a strategy to realise this.

... Optimistic Concurrency Control

Under OCC, transactions have three distinct phases:

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

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



... Optimistic Concurrency Control

Data structures needed for validation:

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

Use the V timestamps as ordering for transactions

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

... Optimistic Concurrency Control

88/136

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

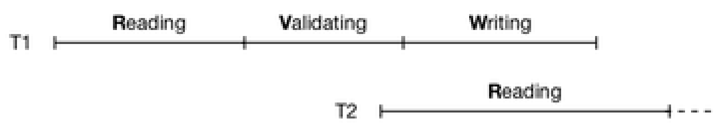
Case 0: serial execution ... no problem

... Optimistic Concurrency Control

89/136

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

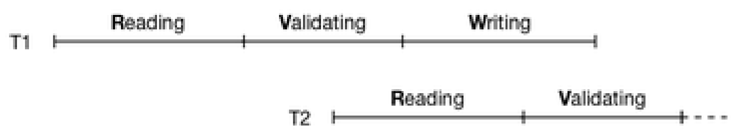


... Optimistic Concurrency Control

90/136

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

91/136

Validation check for transaction T

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

- if $T \in V$ & $V(T_i) < V(T) < F(T_i)$, then check $WS(T_i) \cap WS(T)$ is empty

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

... Optimistic Concurrency Control

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

Problems with OCC:

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

** "Roll back" is relatively cheap

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

Multi-version Concurrency Control

Multi-version concurrency control (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

WTS = timestamp of tx that wrote this data item

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

When a reader T_i is accessing the database

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

... Multi-version Concurrency Control

When a writer T_i attempts to change a data item

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

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

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

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

... Multi-version Concurrency Control

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 PostgreSQL

PostgreSQL uses two styles of concurrency control:

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

From the SQL (PLpgSQL) level:

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

... Concurrency Control in PostgreSQL

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

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

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

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

Exercise 7: Locking in PostgreSQL

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

Reminder:

Transactions are *atomic*

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

Transaction effects are *durable*

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

Implementation of atomicity/durability is intertwined.

Durability

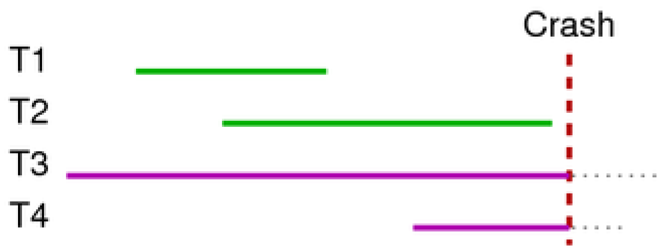
What kinds of "system failures" do we need to deal with?

- single-bit inversion during transfer mem-to-disk
- decay of storage medium on disk (some data changed)
- failure of entire disk device (data 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*

- i.e. `putPage()` and `getPage()` always work as expected

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

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

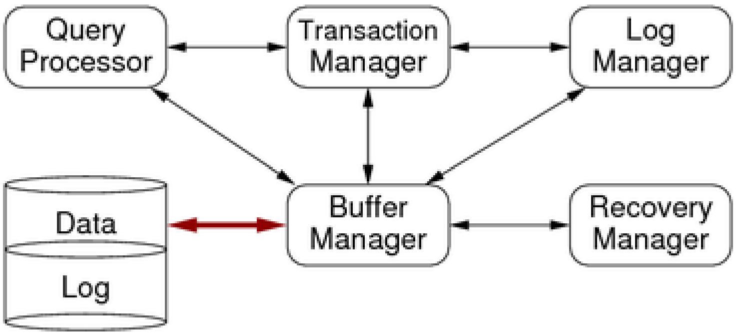
- failure of DBMS processes or operating system
- failure of transactions (`ABORT`)

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

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

PostgreSQL processes make heavy use of shared buffer pool

⇒ transactions do not deal with much local data.

... 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).

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

... 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* ... 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* (PostgreSQL uses WAL)

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:

- 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 (12) completes.

... Undo Logging

122/136

Simplified view of recovery using UNDO logging:

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

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

... Undo Logging

123/136

Algorithmic 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

124/136

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

125/136

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

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

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. 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

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:

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

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

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

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

Simplified view of recovery using UNDO/REDO logging:

- scan log to determine committed/uncommitted txs
- for each uncommitted tx T add <ABORT T> to log
- scan *backwards* through log
  - if <T,X,v,w> and T is not committed, set X to v on disk
- scan *forwards* through log
  - if <T,X,v,w> and T is committed, set X to w on disk

... 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

Recovery in PostgreSQL

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

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: 15 Apr 2020