Week 12

Assignment 2

How robust do I need to make it?

- assume I won't be giving "nasty" inputs (e.g. no `???`)
- need to check appropriate number of items in tuples/queries

How do I know it's correct?

- work out manually what you expect to see
- run your code with diagnostic output to check
- e.g. is it generating the correct MA hash?
    - display the individual hashes, CV, MA hash
    - using hashes + CV, compute the expected MA hash
    - compare observed against expected

Query Processing So Far

Steps in processing an SQL statement

- parse, map to relation algebra (RA) expression
- transform to more efficient RA expression
- instantiate RA operators to DBMS operations
- execute DBMS operations (aka query plan)

Cost-based optimisation:

- generate possible query plans (via heuristics)
- estimate cost of each plan (sum costs of operations)
- choose the lowest-cost plan (... and choose quickly)

Estimating Selection Result Size

Analysis relies on operation and data distribution:

E.g. `select * from R where a = k;`

Case 1: $uniq(R.a)$ ⇒ 0 or 1 result

Case 2: r_R tuples && $size(dom(R.a)) = n$ ⇒ r_R / n results

E.g. `select * from R where a < k;`

Case 1: $k \leq min(R.a)$ ⇒ 0 results

Case 2: $k > max(R.a)$ ⇒ $\cong r_R$ results

Case 3: $size(dom(R.a)) = n$ ⇒ ? $min(R.a)$ ... k ... $max(R.a)$?

Estimating Join Result Size

Analysis relies on semantic knowledge about data/relations.

Consider equijoin on common attr: $R \bowtie_a S$

Case 1: $values(R.a) \cap values(S.a) = \{\} \;\Rightarrow\; size(R \bowtie_a S) = 0$

Case 2: $uniq(R.a)$ and $uniq(S.a) \;\Rightarrow\; size(R \bowtie_a S) \leq min(|R|, |S|)$

Case 3: $pkey(R.a)$ and $fkey(S.a) \;\Rightarrow\; size(R \bowtie_a S) \leq |S|$

Exercise 1: Join Size Estimation

How many tuples are in the output from:

1. `select * from R, S where R.s = S.id`
   where `S.id` is a primary key and `R.s` is a foreign key referencing `S.id`
2. `select * from R, S where R.s <> S.id`
   where `S.id` is a primary key and `R.s` is a foreign key referencing `S.id`
3. `select * from R, S where R.x = S.y`
   where `R.x` and `S.y` have no connection except that *dom(R.x)=dom(S.y)*

Under what conditions will the first query have maximum size?

Cost Estimation: Postscript

Inaccurate cost estimation can lead to poor evaluation plans.

Above methods can (sometimes) give inaccurate estimates.

To get more accurate cost estimates:

- more time ... complex computation of selectivity
- more space ... storage for histograms of data values

Either way, optimisation process costs more (more than query?)

Trade-off between optimiser performance and query performance.

PostgreSQL Query Optimiser

Overview of QOpt Process

Input: tree of **`Query`** nodes returned by parser

Output: tree of **`Plan`** nodes used by query *executor*

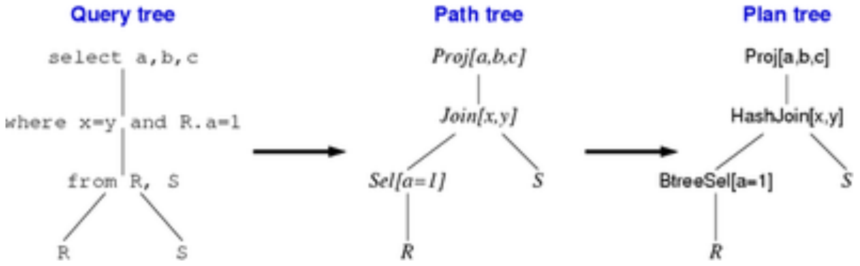
- wrapped in a **`PlannedStmt`** node containing state info

Intermediate data structures are trees of **`Path`** nodes

- a path tree represents one evaluation order for a query

All **`Node`** types are defined in **`include/nodes/*.h`**

... Overview of QOpt Process



QOpt Data Structures

Generic **`Path`** node structure:

```
typedef struct Path
{
   NodeTag     type;           /* scan/join/... */
   NodeTag     pathtype;       /* specific method */
   RelOptInfo *parent;         /* output relation */
   /* estimated execution costs for path */
   Cost        startup_cost;   /* setup cost */
   Cost        total_cost;     /* total cost */
   List       *pathkeys;       /* sort order */
} Path;
```

... QOpt Data Structures

Specialised **`Path`** nodes:

```
typedef struct IndexPath
{
   Path    path;
   List   *indexinfo;     /* physical info on indexes */
   List   *indexclauses; /* index select conditions */
   ...
   double  rows;          /* estimated #results */
} IndexPath;

typedef struct JoinPath
{
   Path      path;
   JoinType  jointype;    /* inner/outer/semi/anti */
   Path     *outerpath;   /* outer part of the join */
   Path     *innerpath;   /* inner part of the join */
   List     *restrictinfo; /* where/join conds */
} JoinPath;
```

Query Optimisation Process

Query optimisation proceeds in two stages (after parsing)...

Rewriting:

- uses PostgreSQL's *rule* system

- query tree is expanded to include e.g. view definitions

Planning and optimisation:

- using cost-based analysis of generated paths
- via one of *two* different path generators
- chooses least-cost path from all those considered

Then produces a **`Plan`** tree from the selected path.

Top-down Trace of QOpt

Top-level of query execution: **`backend/tcop/postgres.c`**

```
exec_simple_query(const char *query_string)
{
   // lots of setting up ... including starting xact
   parsetree_list = pg_parse_query(query_string);
   foreach(parsetree_item, parsetree_list) {
      // Query optimisation
      querytree_list = pg_analyze_and_rewrite(parsetree,...);
      plantree_list = pg_plan_queries(querytree_list,...);
      // Query execution
      portal = CreatePortal(...plantree_list...);
      PortalRun(portal,...);
   }
   // lots of cleaning up ... including close xact
}
```

Assumes that we are dealing with multiple queries (i.e. SQL statements)

... Top-down Trace of QOpt

`pg_analyze_and_rewrite()`

- take a parse tree (from SQL parser)
- transforms Parse tree into Query tree (SQL → RA)
- applies rewriting rules (e.g. views)
- returns a list of Query trees

Code in: **`backend/tcop/postgres.c`**

... Top-down Trace of QOpt

`pg_plan_queries()`

- takes a list of parsed/re-written queries
- plans each one via **`planner()`**
    - which invokes **`subquery_planner()`** on each query
- returns a list of query plans

Code in: **`backend/optimizer/plan/planner.c`**

... Top-down Trace of QOpt

`subquery_planner()`

- performs algebraic transformations/simplifications, e.g.
    - simplifies conditions in **where** clauses
    - converts sub-queries in **where** to top-level join
    - moves **having** clauses with no aggregate into **where**
    - flattens sub-queries in join list
    - simplifies join tree (e.g. removes redundant terms), etc.
- sets up canonical version of query for plan generation
- invokes **`grouping_planner()`** to produce best path

Code in: **`backend/optimizer/plan/planner.c`**

... Top-down Trace of QOpt

`grouping_planner()` produces plan for one SQL statement

- preprocesses target list for INSERT/UPDATE
- handles "planning" for extended-RA SQL constructs:
    - set operations: UNION/INTERSECT/EXCEPT
    - GROUP BY, HAVING, aggregations
    - ORDER BY, DISTINCT, LIMIT
- invokes **`query_planner()`** for select/join trees

Code in: **`backend/optimizer/plan/planmain.c`**

... Top-down Trace of QOpt

`query_planner()` produces plan for a select/join tree

- make list of tables used in query
- split **where** qualifiers ("quals") into
    - restrictions (e.g. `r.a=1`) ... for selections
    - joins (e.g. `s.id=r.s`) ... for joins
- search for quals to enable merge/hash joins

- invoke **`make_one_rel()`** to find best path/plan

Code in: **`backend/optimizer/plan/planmain.c`**

... Top-down Trace of QOpt

`make_one_rel()` generates possible plans, selects best

- generate scan and index paths for base tables
    - using of restrictions list generated above
- generate access paths for the entire join tree
    - recursive process, controlled by **`make_rel_from_joinlist()`**
- returns a single "relation", representing result set

Code in: **`backend/optimizer/path/allpaths.c`**

Join-tree Generation

`make_rel_from_joinlist()` arranges path generation

- switches between two possible path tree generators
- path tree generators finally return best cost path

Standard path tree generator (`standard_join_search()`):

- "exhaustively" generates join trees (a la System R)
- starts with 2-way joins, finds best combination
- then adds extra table to give 3-table join, etc.

Code in: `backend/optimizer/path/{allpaths.c,joinrels.c}`

... Join-tree Generation

Genetic query optimiser (`geqo`):

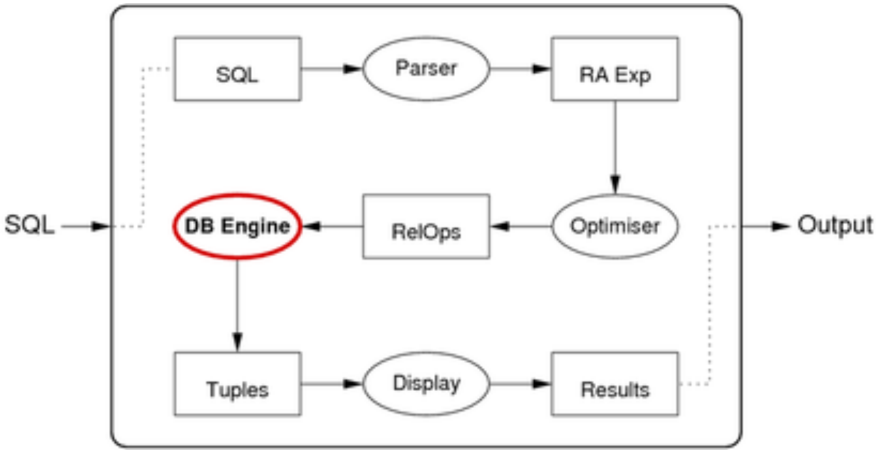
- uses genetic algorithm (GA) to generate path trees
- based on GA designed for "travelling salesman" problem
- goals of this approach:
  - find near-optimal solution
  - examine far less than entire search space
- used as path generator in PostgreSQL for large joins
- threshold determined by `geqo_threshold` config param

Code in: `backend/optimizer/geqo/*.c`

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

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

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:

- blocks execute "concurrently" as producer/consumer pairs
- first operator acts as producer; second as consumer
- 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

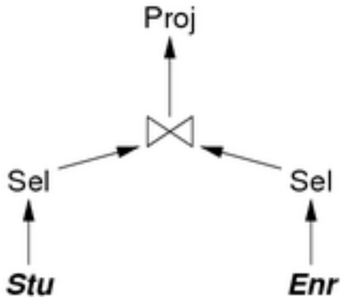
Consider the query:

```
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';
```

which maps to the RA expression

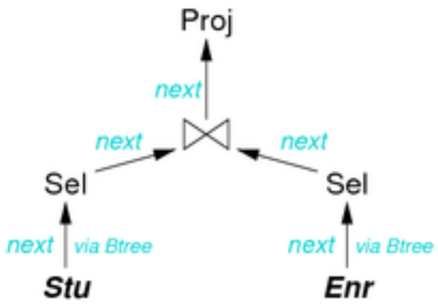
$$Proj_{[id,course,mark]}(Join_{[student=id]}(Sel_{[05s2]}(Enr),Sel_{[John]}(Stu)))$$

which could represented by the RA expression tree



... Pipelining Example

Modelled as communication between RA tree nodes:



Note: likely that projection is combined with join in real DBMSs.

... Pipelining Example

This query might be executed as

```
System:
    iter0 = startScan(Result)
    while (Tup = nextTuple(iter0)) { display Tup }
    endScan(iter0)
Result:
    iter1 = startScan(Join)
    while (T = nextTuple(iter1))
        { T' = project(T); return T' }
    endScan(iter1)
Sel1:
    iter4 = startScan(Btree(Enrolment,'semester=05s2'))
    while (A = nextTuple(iter4)) { return A }
    endScan(iter4)
...
```

... Pipelining Example

```
...
Join: -- nested-loop join
    iter2 = startScan(Sel1)
    while (R = nextTuple(iter2) {
        iter3 = startScan(Sel2)
        while (S = nextTuple(iter3))
            { if (matches(R,S) return (RS) }
        endScan(iter3) // better to reset(iter3)
    }
    endScan(iter2)
Sel2:
    iter5 = startScan(Btree(Student,'name=John'))
    while (B = nextTuple(iter5)) { return B }
    endScan(iter5)
```

Disk Accesses

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

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

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

PostgreSQL uses pipelining ...

- 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

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`**

... PostgreSQL Executor

Much simplified view of PostgreSQL executor:

```
ExecutePlan(execState, planStateNode, ...) {
   process "before each statement" triggers
   for (;;) {
      tuple = ExecProcNode(planStateNode)
      if (no more tuples) return END
      check tuple validity // MVCC
      if (got a tuple) break
   }
   process "after each statement" triggers
   return tuple
}
...
```

... PostgreSQL Executor

Executor overview (cont):

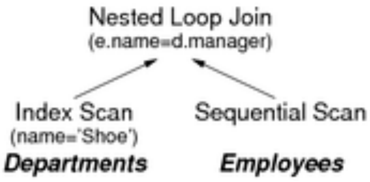
```
...
ExecProcNode(node) {
   switch (nodeType(node)) {
   case SeqScan:
      result = ExecSeqScan(node); break;
   case NestLoop:
      result = ExecNestLoop(node); break;
   ...
   }
   return result;
}
```

Example PostgreSQL Execution

Consider the query:

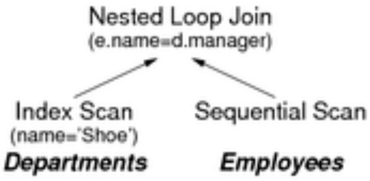
```
-- get manager's age and # employees in Shoe department
select e.age, d.nemps
from   Departments d, Employees e
where  e.name = d.manager and d.name ='Shoe'
```

and its execution plan tree



... Example PostgreSQL Execution

The execution plan tree



contains three nodes:

- `NestedLoop` with join condition (Outer.manager = Inner.name)
- `IndexScan` on Departments with selection (name = 'Shoe')
- `SeqScan` on Employees

... 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

Execution: `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 perform "better"?

Good 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. 90% withdrawals; 10% deposits; 50% 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

Example: Select on non-indexed attribute

```
uni=# explain
uni=# select * from Students where stype='local';
                     QUERY PLAN
----------------------------------------------------
 Seq Scan on students
             (cost=0.00..556.10 rows=20073 width=9)
   Filter: ((stype)::text = 'local'::text)


uni=# explain analyze
uni=# select * from Students where stype='local';
                       QUERY PLAN
----------------------------------------------------------
 Seq Scan on students
             (cost=0.00..556.10 rows=20073 width=9)
             (actual time=0.027..4.529 rows=20048 loops=1)
   Filter: ((stype)::text = 'local'::text)
   Rows Removed by Filter: 11000
 Total runtime: 5.4 ms
```

... EXPLAIN Examples

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

```
uni=# explain
uni-# select s.sid,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 non-indexed attribute

```
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
```

Exercise 2: EXPLAIN examples

Using the following database ...

People(**id**, family, given, birthday, ...)
Courses(**id**, *subject*, term, ...)
Subjects(**id**, code, title, ...)
CourseEnrolments(*student*, *course*, grade, mark, ...)

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

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 s.family 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

Transaction: application-level operation requiring multiple DB operations

Data integrity is assured if transactions satisfy the following:

Atomicity

- Either all operations of a tx appear in database or none do

Consistency

- Execution of a tx in isolation preserves data consistency

Isolation

- Each tx is "unaware" of other concurrent tx's

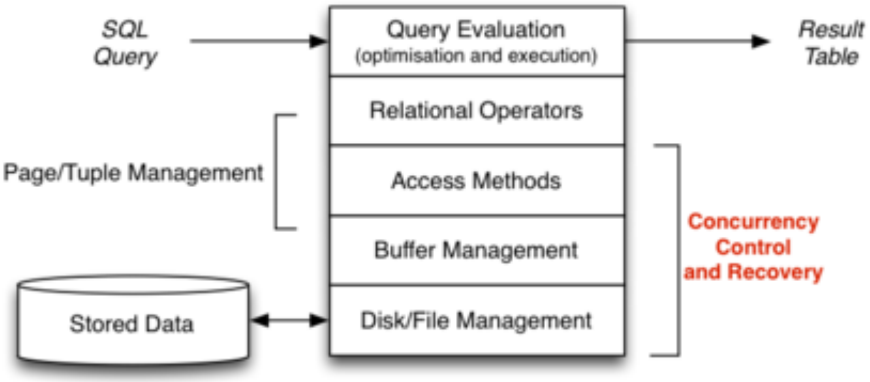
Durability

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

---

... Transaction Processing

Where transaction processing fits in the DBMS:



Schedules

A *schedule* gives the sequence of operations from ≥ *1* tx

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

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

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

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

- operations from individual T_i's are interleaved

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

Transaction Anomalies

What problems can occur with uncontrolled 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

Example of Transaction Failure

Above examples 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 rollback might occur:

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

... Example of 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

Produced: 17 May 2016