Week 08 Lectures

Assignment 2

Aim: experimental analysis of signature-based indexing

- tuple-level superimposed codeword signatures
- page-level superimposed codeword signatures
- bit-sliced superimposed codeword signatures

Large numbers of tuples, inserted into a *relation*:

- implemented as one data file plus three signature files

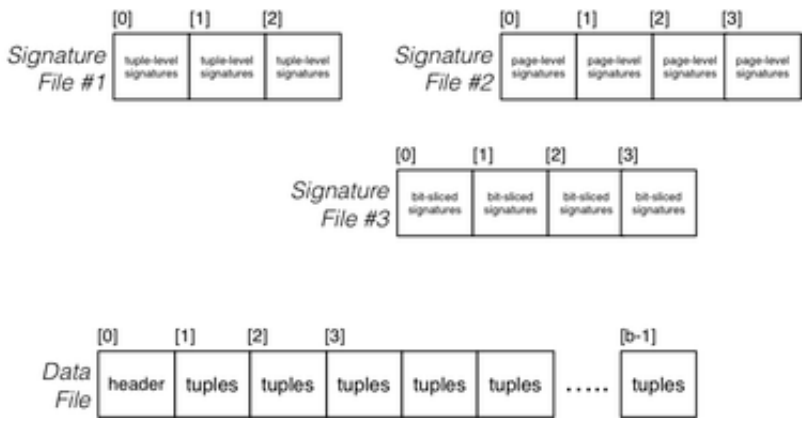
Produce several instance of (data+signatures) relations

Run a set of benchmark PMR queries on these relations

Measure query costs and analyse

... Assignment 2

File structures:



... Assignment 2

We supply:

- a program to generate pseudo-random tuples (lots of them)
    - where tuples look like: (id, name, *numeric-attributes*)
- a hash function (from PostgreSQL)

You write:

- a program to build (data+signature) files from tuples
- a program to run queries (val,?,?,val,?,...) against the data

Your programs must be parameterized:

- builder accepts different values for signature parameters
- query processor specifies what kind of signatures to use

... Assignment 2

Experiments:

- run each query using each signature type
- for each sig type
  - read appropriate signature pages (maybe all)
  - determine which data pages to read
  - fetch pages and search for matching tuples
  - determine total cost (#signature pages + #data pages)
- record costs and analyse/compare

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 rewriting/heuristics)
- estimate cost of each plan (sum costs of operations)
- choose the lowest-cost plan (... and choose quickly)

Expression Rewriting Rules

Since RA is a well-defined formal system

- there exist many algebraic laws on RA expressions
- which can be used as a basis for expression rewriting
- in order to produce *equivalent* (more-efficient?) expressions

Expression transformation based on such rules can be used

- to simplify/improve SQL→RA mapping results
- to generate new plan variations to check in query optimisation

Relational Algebra Laws

Commutative and Associative Laws:

- $R \bowtie S \leftrightarrow S \bowtie R$, $(R \bowtie S) \bowtie T \leftrightarrow R \bowtie (S \bowtie T)$ (natural join)
- $R \cup S \leftrightarrow S \cup R$, $(R \cup S) \cup T \leftrightarrow R \cup (S \cup T)$
- $R \bowtie_{Cond} S \leftrightarrow S \bowtie_{Cond} R$ (theta join)
- $\sigma_c (\sigma_d (R)) \leftrightarrow \sigma_d (\sigma_c (R))$

Selection splitting (where *c* and *d* are conditions):

- $\sigma_{c \land d}(R) \leftrightarrow \sigma_c (\sigma_d (R))$
- $\sigma_{c \lor d}(R) \leftrightarrow \sigma_c(R) \cup \sigma_d(R)$

... Relational Algebra Laws

Selection pushing ($\sigma_c(R \cup S)$ and $\sigma_c(R \cup S)$):

- $\sigma_c(R \cup S) \leftrightarrow \sigma_c R \cup \sigma_c S$, $\sigma_c(R \cap S) \leftrightarrow \sigma_c R \cap \sigma_c S$

Selection pushing with join ...

- $\sigma_c (R \bowtie S) \leftrightarrow \sigma_c(R) \bowtie S$ (if *c* refers only to attributes from *R*)
- $\sigma_c (R \bowtie S) \leftrightarrow R \bowtie \sigma_c(S)$ (if *c* refers only to attributes from *S*)

If *condition* contains attributes from both *R* and *S*:

- $\sigma_{c' \wedge c''} (R \bowtie S) \leftrightarrow \sigma_{c'}(R) \bowtie \sigma_{c''}(S)$
- c' contains only *R* attributes, c'' contains only *S* attributes

... Relational Algebra Laws

Rewrite rules for projection ...

All but last projection can be ignored:

- $\pi_{L1} (\pi_{L2} (... \pi_{Ln} (R))) \rightarrow \pi_{L1} (R)$

Projections can be pushed into joins:

- $\pi_L (R \bowtie_c S) \leftrightarrow \pi_L (\pi_M(R) \bowtie_c \pi_N(S))$

where

- *M* and *N* must contain all attributes needed for *c*
- *M* and *N* must contain all attributes used in *L* ($L \subset M \cup N$)

... Relational Algebra Laws

Subqueries ⇒ convert to a join

Example: (on schema Courses(id,code,...), Enrolments(cid,sid,...), Students(id,name,...)

```
select c.code, count(*)
from   Courses c
where  c.id in (select cid from Enrolments)
group  by c.code
```

becomes

```
select c.code, count(*)
from   Courses c join Enrolments e on c.id = e.cid
group  by c.code
```

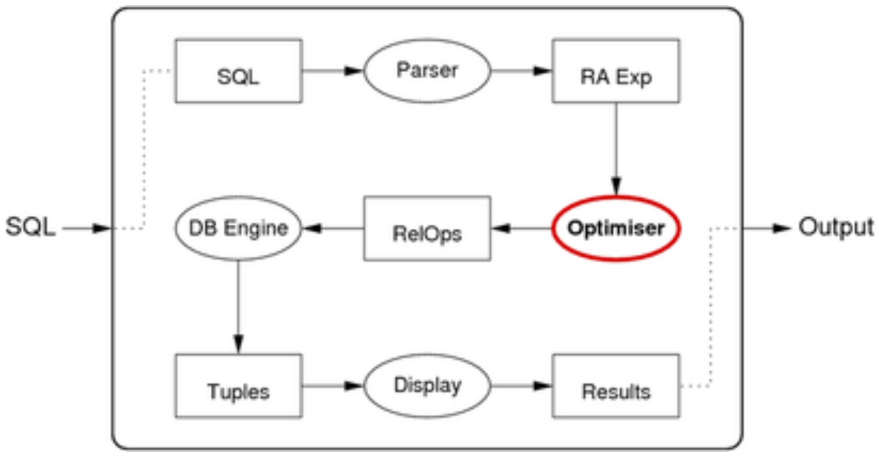
but not

```
select e.sid as student_id, e.cid as course_id
from   Enrolments e
where  e.sid = (select max(id) from Students)
```

Query Optimisation

Query Optimisation

Query optimiser: RA expression → efficient evaluation plan



... Query Optimisation

Query optimisation is a critical step in query evaluation.

The query optimiser

- takes relational algebra expression from SQL compiler
- produces sequence of RelOps to evaluate the expression
- *query execution plan* should provide efficient evaluation

"Optimisation" is a misnomer since query optimisers

- aim to find a good plan ... but maybe not optimal

Observed Query Time = Planning time + Evaluation time

... Query Optimisation

Why do we not generate optimal query execution plans?

Finding an optimal query plan ...

- requires exhaustive search of a *space of possible plans*
- for each possible plan, need to estimate cost (not cheap)

Even for relatively small query, search space is *very large*.

Compromise:

- do limited search of query plan space (guided by heuristics)
- *quickly* choose a *reasonably efficient* execution plan

Approaches to Optimisation

Three main classes of techniques developed:

- algebraic (equivalences, rewriting, heuristics)
- physical (execution costs, search-based)
- semantic (application properties, heuristics)

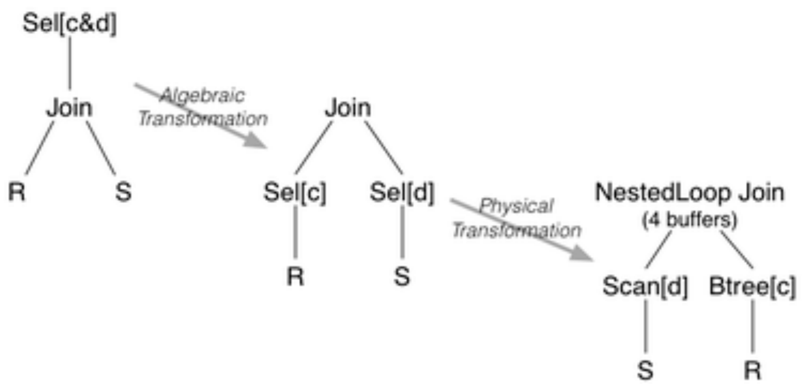
All driven by aim of minimising (or at least reducing) "cost".

Real query optimisers use a combination of algrebraic+physical.

Semantic QO is good idea, but expensive/difficult to implement.

... Approaches to Optimisation

Example of optimisation transformations:



For join, may also consider sort/merge join and hash join.

Cost-based Query Optimiser

Approximate algorithm for cost-based optimisation:

```
translate SQL query to RAexp
for enough transformations RA' of RAexp {
   while (more choices for RelOps) {
      plan = {}; i = 0
      for each node e of RA' (recursively) {
         select RelOp method for e
         plan[i++] = RelOp method for e
      }
      cost = 0
      for each op in plan[] { cost += Cost(op) }
      if (cost < MinCost)
         { MinCost = cost;  BestPlan = plan }
   }
}
```

Heuristics: push selections down, consider only left-deep join trees.

Exercise 1: Alternative Join Plans

Consider the schema

```
Students(id,name,....)   Enrol(student,course,mark)
Staff(id,name,...)    Courses(id,code,term,lic,...)
```

the following query on this schema

```
select c.code, s.id, s.name
from   Students s, Enrol e, Courses c, Staff f
where  s.id=e.student and e.course=c.id
       and c.lic=f.id and c.term='11s2'
       and f.name='John Shepherd'
```

Show some possible evaluation orders for this query.

Cost Models and Analysis

The cost of evaluating a query is determined by:

- size of relations (database relations and temporary relations)
- access mechanisms (indexing, hashing, sorting, join algorithms)
- size/number of main memory buffers (and replacement strategy)

Analysis of costs involves *estimating*:

- size of intermediate results
- number of secondary storage accesses

Choosing Access Methods (RelOps)

Performed for each node in RA expression tree ...

Inputs:

- a single RA operation (σ, π, $\bowtie$)
- information about file organisation, data distribution, ...
- list of operations available in the database engine

Output:

- specific DBMS operation to implement this RA operation

... Choosing Access Methods (RelOps)

Example:

- RA operation: $Sel_{[name='John' \wedge age>21]}(Student)$
- `Student` relation has B-tree index on `name`
- database engine (obviously) has B-tree search method

giving

```
tmp[i]   := BtreeSearch[name='John'](Student)
tmp[i+1] := LinearSearch[age>21](tmp[i])
```

Where possible, use pipelining to avoid storing `tmp[i]` on disk.

... Choosing Access Methods (RelOps)

Rules for choosing σ access methods:

- $\sigma_{A=c}(R)$ and `R` has index on `A` ⇒ `indexSearch[A=c](R)`
- $\sigma_{A=c}(R)$ and `R` is hashed on `A` ⇒ `hashSearch[A=c](R)`
- $\sigma_{A=c}(R)$ and `R` is sorted on `A` ⇒ `binarySearch[A=c](R)`
- $\sigma_{A \geq c}(R)$ and `R` has clustered index on `A`
  ⇒ `indexSearch[A=c](R)` then scan
- $\sigma_{A \geq c}(R)$ and `R` is hashed on `A`
  ⇒ `linearSearch[A>=c](R)`

... Choosing Access Methods (RelOps)

Rules for choosing $\bowtie$ access methods:

- $R \bowtie S$ and `R` fits in memory buffers ⇒ `bnlJoin(R,S)`
- $R \bowtie S$ and `S` fits in memory buffers ⇒ `bnlJoin(S,R)`
- $R \bowtie S$ and `R`,`S` sorted on join attr ⇒ `smJoin(R,S)`
- $R \bowtie S$ and `R` has index on join attr ⇒ `inlJoin(S,R)`
- $R \bowtie S$ and no indexes, no sorting ⇒ `hashJoin(R,S)`

(`bnl` = block nested loop; `inl` = index nested loop; `sm` = sort merge)

Cost Estimation

Without executing a plan, cannot always know its precise cost.

Thus, query optimisers *estimate* costs via:

- cost of performing operation (dealt with in earlier lectures)
- size of result (which affects cost of performing next operation)

Result size determined by statistical measures on relations, e.g.

r_S	cardinality of relation *S*
R_S	avg size of tuple in relation *S*
$V(A,S)$	# distinct values of attribute *A*
$min(A,S)$	min value of attribute *A*
$max(A,S)$	max value of attribute *A*

Estimating Projection Result Size

Straightforward, since we know:

- number of tuples in output

  $r_{out} = |\ \pi_{a,b,..}(T)\ | = |\ T\ | = r_T$ (in SQL, because of bag semantics)

- size of tuples in output

  R_{out} = sizeof(a) + sizeof(b) + ... + tuple-overhead

Assume page size B, $b_{out} = \lceil r_T / c_{out} \rceil$, where $c_{out} = \lfloor B/R_{out} \rfloor$

If using `select distinct` ...

- $|\ \pi_{a,b,..}(T)\ |$ depends on proportion of duplicates produced

---

Estimating Selection Result Size

26/141

Selectivity = fraction of tuples expected to satisfy a condition.

Common assumption: attribute values uniformly distributed.

Example: Consider the query

```
select * from Parts where colour='Red'
```

If $V(colour,Parts)=4$, $r=1000$ $\Rightarrow$ $|\sigma_{colour=red}(Parts)|=250$

In general, $|\ \sigma_{A=c}(R)\ | \ \cong\ r_R / V(A,R)$

Heuristic used by PostgreSQL: $|\ \sigma_{A=c}(R)\ | \ \cong\ r/10$

... Estimating Selection Result Size

27/141

Estimating size of result for e.g.

```
select * from Enrolment where year > 2005;
```

Could estimate by using:

- uniform distribution assumption, r, min/max years

Assume: min(year)=2000, max(year)=2009, $|Enrolment|=10^5$

- 10^5 from 2000-2009 means approx 10000 enrolments/year
- this suggests 40000 enrolments since 2006

Heuristic used by some systems: $|\ \sigma_{A>c}(R)\ | \cong r/3$

... Estimating Selection Result Size

28/141

Estimating size of result for e.g.

```
select * from Enrolment where course <> 'COMP9315';
```

Could estimate by using:

- uniform distribution assumption, r, domain size

e.g. $|\ V(course,Enrolment)\ |$ = 2000, $|\ \sigma_{A<>c}(E)\ | = r * 1999/2000$

Heuristic used by some systems: $| \sigma_{A<>c}(R) | \cong r$

Exercise 2: Selection Size Estimation

Assuming that

- all attributes have uniform distribution of data values
- attributes are independent of each other

Give formulae for the number of expected results for

1. `select * from R where not A=k`
2. `select * from R where A=k and B=j`
3. `select * from R where A in (k,l,m,n)`

where *j*, *k*, *l*, *m*, *n* are constants.

Assume: $V(A,R) = 10$ and $V(B,R)=100$ and $r=1000$

... Estimating Selection Result Size

How to handle non-uniform attribute value distributions?

- collect statistics about the values stored in the attribute/relation
- store these as e.g. a histogram in the meta-data for the relation

So, for part colour example, might have distribution like:

`White`: 35% `Red`: 30% `Blue`: 25% `Silver`: 10%

Use histogram as basis for determining # selected tuples.

Disadvantage: cost of storing/maintaining histograms.

... Estimating Selection Result Size

Summary: 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)* $\Rightarrow size(R \bowtie_a S) \le min(|R|, |S|)$

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

Exercise 3: 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*

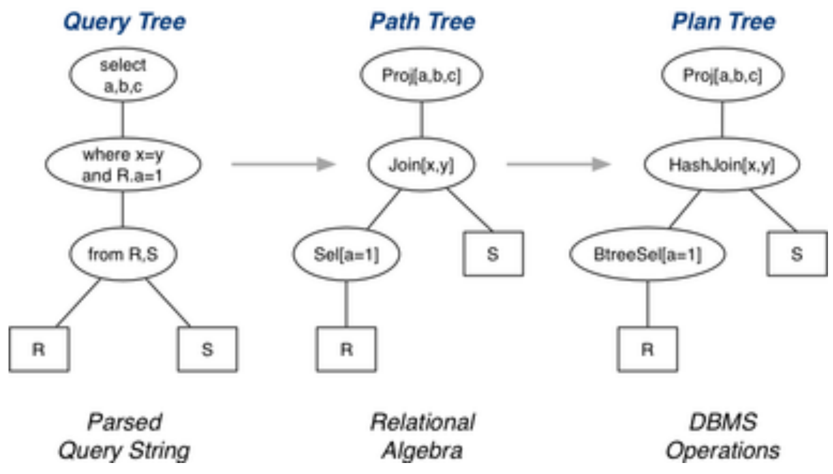
- wrapped in a **`PlannedStmt`** node containing state info

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
   PathTarget *pathtarget;    // list of Vars/Exprs, width
   // estimated execution costs for path ...
   Cost        startup_cost;  // setup cost
   Cost        total_cost;    // total cost
   List       *pathkeys;      // sort order
} Path;
```

`PathKey` = (opfamily:Oid, strategy:SortDir, nulls_first:bool)

... QOpt Data Structures

Specialised `Path` nodes (simplified):

```
typedef struct IndexPath
{
   Path    path;
   IndexOptInfo *indexinfo; // index for scan
   List   *indexclauses; // index select conditions
   ...
   ScanDirection  indexscandir; // used by planner
   Selectivity  indexselectivity; // 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; // join condition(s)
} 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, 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 (like 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
- handles joins involving > `geqo_threshold` relations
- goals of this approach:
    - find near-optimal solution
    - examine far less than entire search space

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

... Join-tree Generation

Basic idea of genetic algorithm:

```
Optimize(join)
{
   t = 0
   p = initialState(join)  // pool of (random) join orders
   for (t = 0; t < #generations; t++) {
      p' = recombination(p) // get parts of good join orders
      p'' = mutation(p')    // generate new variations
      p = selection(p'',p)  // choose best join orders
   }
}
```

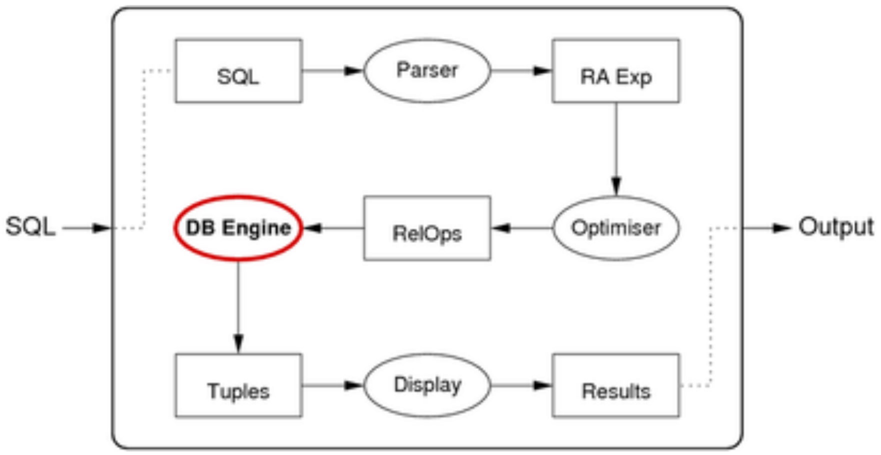
#generations determined by size of initial pool of join orders

---

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

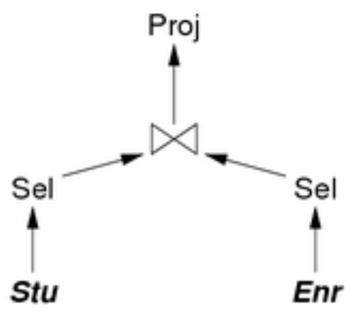
Consider the query:

```
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

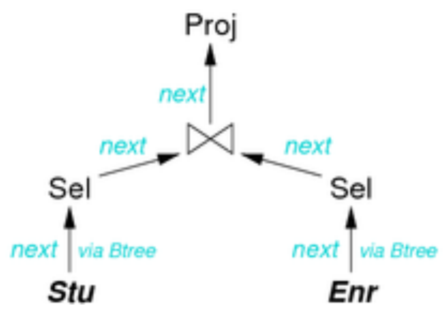
$$Proj_{[id,course,mark]}(Join_{[student=id]}(Sel_{[05s2]}(Enr),Sel_{[John]}(Stu)))$$

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.

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;
}
```

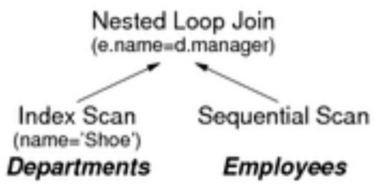
Example PostgreSQL Execution

Consider the query:

```
-- get manager's age and # employees in Shoe department
select e.age, d.nemps
```

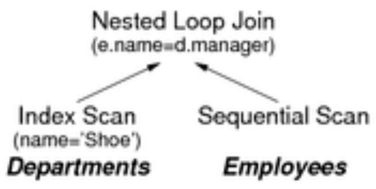
```
from   Departments d, Employees e
where  e.name = d.manager and d.name ='Shoe'
```

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

Database

```
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, ...)
```

where

```
       table_name           | n_records
---------------------------+-----------
 course_enrolments          |    525688
 courses                    |     73220
 people                     |     55767
 program_enrolments         |    193456
 students                   |     31361
 subjects                   |     17779
```

... 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

78/141

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

79/141

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

80/141

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

81/141

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 4: 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

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

Concurrent transactions are

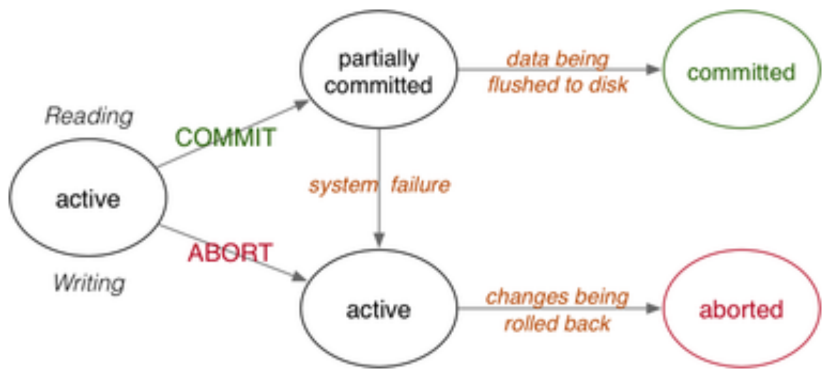
- desirable, for improved performance
- problematic, because of potential unwanted interactions

To ensure problem-free concurrent transactions:

- **A**tomic ... whole effect of tx, or nothing
- **C**onsistent ... individual tx's are "correct" (wrt application)
- **I**solated ... each tx behaves as if no concurrency
- **D**urable ... effects of committed tx's persist

... Transaction Processing

Transaction states:



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

... Transaction Processing

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

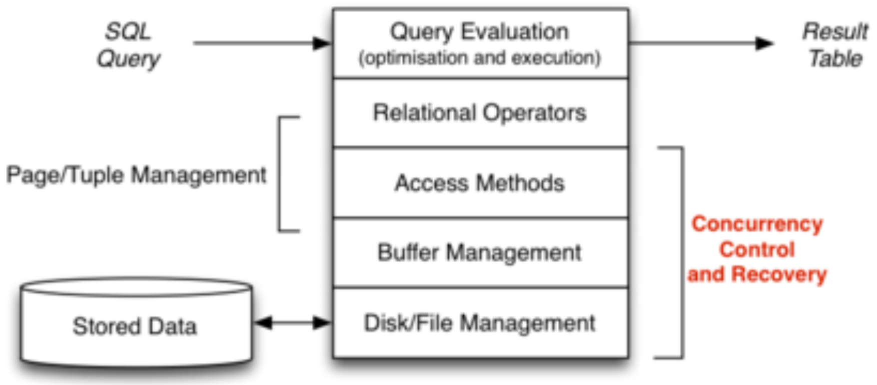
- redundancy, to deal with hardware failures
- logging/checkpoint mechanisms, to recover state

Isolation is handled by *concurrency control* mechanisms

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

... Transaction Processing

Where transaction processing fits in the DBMS:



Transaction Terminology

94/141

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

95/141

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)

... Transaction Terminology

96/141

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

... 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 operations arbitrarily

- and may produce a database that is not consistent
- after all of the transactions have committed

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

... 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

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

Transaction Isolation

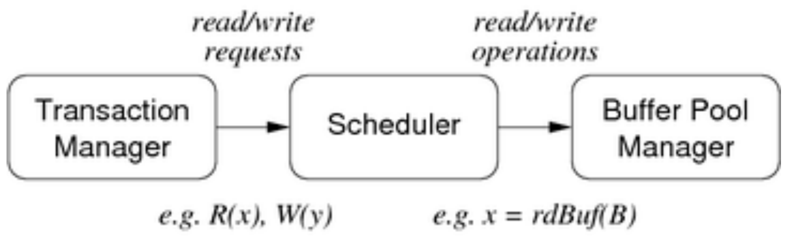
Transaction Isolation

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

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

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	Dirty	Nonrepeatable	Phantom

Level	Read	Read	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

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`

... Transaction Isolation Levels

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.

... Transaction Isolation Levels

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

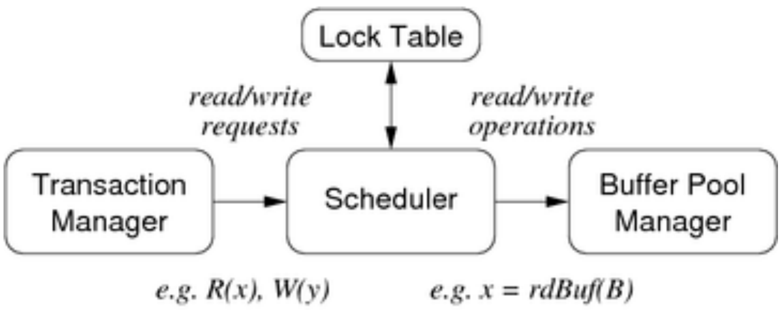
Concurrency Control

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

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

Synchronise access to shared data items via following rules:

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

These rules alone do not guarantee serializability.

... Lock-based Concurrency Control

Consider the following schedule, using locks:

```
T1(a): Lr(Y)     R(Y)          continued
T2(a):     Lr(X)    R(X) U(X)  continued

T1(b):     U(Y)         Lw(X) W(X) U(X)
T2(b): Lw(Y)....W(Y) U(Y)
```

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

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

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

Two-Phase Locking

To guarantee serializability, we require an additional constraint:

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

Each transaction is then structured as:

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

Clearly, this reduces potential concurrency ...

Problems with Locking

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 "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

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 ...

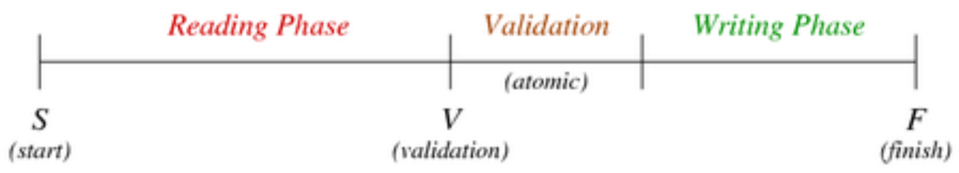
- don't lock (allow arbitrary interleaving of operations)
- check just before commit that no conflicts occurred
- if problems, roll back conflicting transactions

... Optimistic Concurrency Control

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

Data structures needed for validation:

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

Use the *V* timestamps as ordering for transactions

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

... Optimistic Concurrency Control

Validation check for transaction *T*

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

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

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

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

---

Multi-version Concurrency Control

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 last writer; RTS = timestamp of last reader

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

When a reader T_i is accessing the database

- ignore any data item D created after T_i started (WTS(D) > TS(T_i))
- use only newest version V satisfying WTS(V) < TS(T_i)

When a writer T_j attempts to change a data item

- find newest version V satisfying WTS(V) < TS(T_j)
- if no later versions exist, create new version of data item
- otherwise, abort T_j

... 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 while T_2 deletes then counts; then counts are compared

Use locks if application needs every tx to see same current version

- `LOCK TABLE` locks an entire table
- `SELECT FOR UPDATE` locks only the selected rows

Exercise 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;
```

Produced: 13 Sep 2018