Week 10 Lecture

Assignment 1 Review

1/69

64 submissions ... 8 didn't compile or crashed server

Test data: loaded 69K User tuples, 111K Session tuples

- file size: Users 5MB..38MB, Sessions 8MB..62MB

One tuple with 128-char user and 128-char domain

- 20 submissions stored data incorrectly

Quiz3

2/69

Topic: insertion, searching, hashing

Completed by 101/109 students; average mark 3.6/4

Questions:

1. Binary search ... 82 correct (14+2 or None)
2. One-type query ... 96 correct
3. Linear-hashed file ... 88 correct
4. Static hashing ... 91 correct

Quiz4

3/69

Topic: indexes, multi-attribute hashing

Completed by 104/109 students; average mark 3.6/4

Questions:

1. Multi-attribute hash ... 96 correct
2. Bitmap index ... 98 correct
3. B-tree insertion ... 98 correct (13,25) or (19,25)
4. Index space ... 98 correct

Hash Join

Hash Join

5/69

Basic idea:

- use hashing as a technique to partition relations
- to avoid having to consider all pairs of tuples

Requires sufficent memory buffers

- to hold substantial portions of partitions
- (preferably) to hold largest partition of outer relation

Other issues:

- works only for equijoin `R.i=S.j` (but this is a common case)
- susceptible to data skew (or poor hash function)

Variations: *simple*, *grace*, *hybrid*.

Simple Hash Join

Basic approach:

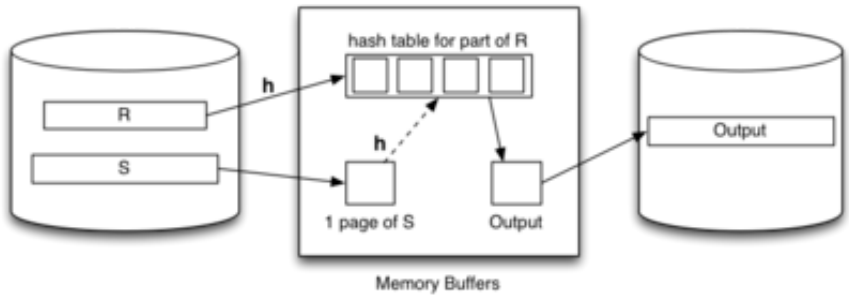
- hash part of outer relation *R* into memory buffers (build)
- scan inner relation *S*, using hash to search (probe)
  - if R.i=S.j, then h(R.i)=h(S.j) (hash to same buffer)
  - only need to check one memory buffer for each *S* tuple
- repeat until whole of *R* has been processed

No overflows allowed in in-memory hash table

- works best with uniform hash function
- can be adversely affected by data/hash skew

... Simple Hash Join

Data flow:



... Simple Hash Join

Algorithm for simple hash join *Join[R.i=S.j](R,S)*:

```
for each tuple r in relation R {
   if (buffer[h(R.i)] is full) {
      for each tuple s in relation S {
         for each tuple rr in buffer[h(S.j)] {
            if ((rr,s) satisfies join condition) {
               add (rr,s) to result
      }  }  }
      clear all hash table buffers
   }
   insert r into buffer[h(R.i)]
}
```

join tests $\leq r_S.c_R$ (cf. nested-loop $r_S.r_R$)

page reads depends on #buffers *N* and properties of data/hash.

Exercise 1: Simple Hash Join Cost

9/69

Consider executing *Join[i=j](R,S)* with the following parameters:

- $r_R = 1000$, $b_R = 50$, $r_S = 3000$, $b_S = 150$, $c_{Res} = 30$
- *R.i* is primary key, each *R* tuple joins with 2 *S* tuples
- DBMS has $N = 43$ buffers available for the join
- data + hash have reasonably uniform distribution

Calculate the cost for evaluating the above join

- using simple hash join
- compute #pages read/written
- compute #join-condition checks performed
- assume that hash table has *L=0.75* for each partition

Grace Hash Join

10/69

Basic approach (for $R \bowtie S$):

- partition both relations on join attribute using hashing (*h1*)
- load each partition of *R* into N-buffer hash table (*h2*)
- scan through corresponding partition of *S* to form results
- repeat until all partitions exhausted

For best-case cost ($O(b_R + b_S)$):

- need $\geq \sqrt{b_R}$ buffers to hold largest partition of outer relation

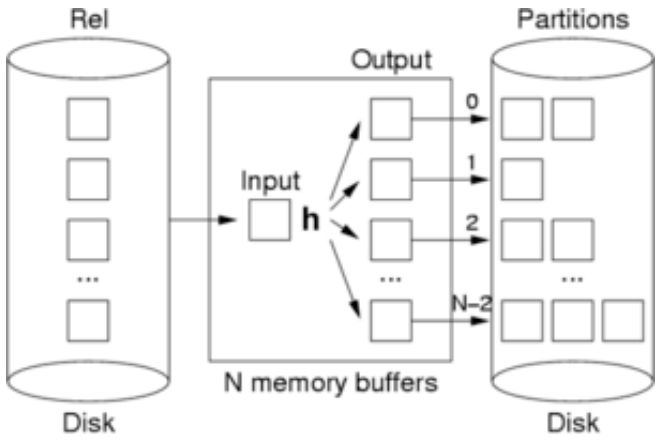
If $< \sqrt{b_R}$ buffers or poor hash distribution

- need to scan some partitions of *S* multiple times

... Grace Hash Join

11/69

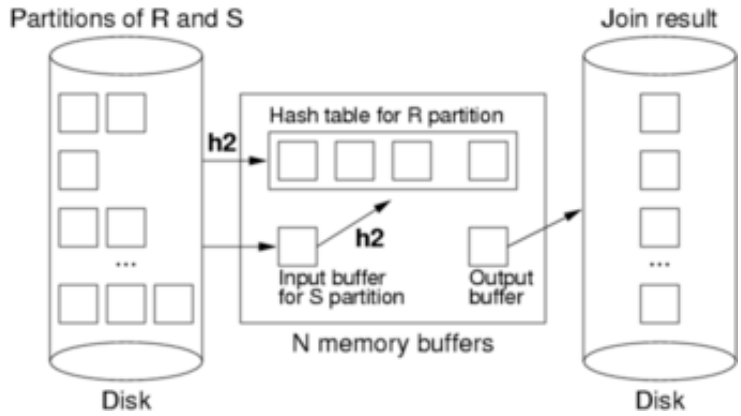
Partition phase (applied to both *R* and *S*):



... Grace Hash Join

12/69

Probe/join phase:



The second hash function (`h2`) simply speeds up the matching process.
Without it, would need to scan entire *R* partition for each record in *S* partition.

... Grace Hash Join

13/69

Cost of grace hash join:

- #pages in all partition files of *Rel* $\cong b_{Rel}$ (maybe slightly more)
- partition relation *R* ... Cost = $b_R.T_r + b_R.T_w = 2b_R$
- partition relation *S* ... Cost = $b_S.T_r + b_S.T_w = 2b_S$
- probe/join requires one scan of each (partitioned) relation
  Cost = $b_R + b_S$
- all hashing and comparison occurs in memory $\Rightarrow$ $\cong 0$ cost

Total Cost = $2b_R + 2b_S + b_R + b_S = 3\,(b_R + b_S)$

Exercise 2: Grace Hash Join Cost

14/69

Consider executing *Join[i=j](R,S)* with the following parameters:

- $r_R = 1000$, $b_R = 50$, $r_S = 3000$, $b_S = 150$, $c_{Res} = 30$
- *R.i* is primary key, each *R* tuple joins with 2 *S* tuples
- DBMS has $N = 42$ buffers available for the join
- data + hash have reasonably uniform distribution

Calculate the cost for evaluating the above join

- using Grace hash join
- compute #pages read/written
- compute #join-condition checks performed
- assume that no *R* partition is larger than 40 pages

Exercise 3: Grace Hash Join Cost

15/69

Consider executing *Join[i=j](R,S)* with the following parameters:

- $r_R = 1000$, $b_R = 50$, $r_S = 3000$, $b_S = 150$, $c_{Res} = 30$
- *R.i* is primary key, each *R* tuple joins with 2 *S* tuples
- DBMS has $N = 42$ buffers available for the join
- data + hash have reasonably uniform distribution

Calculate the cost for evaluating the above join

- using Grace hash join
- compute #pages read/written
- compute #join-condition checks performed
- assume that one *R* partition has 50 pages, others < 40 pages
- assume that the corresponding *S* partition has 30 pages

Hybrid Hash Join

16/69

A variant of grace join if we have $\sqrt{b_R} < N < b_R+2$

- create $k \ll N$ partitions, m in memory, $k-m$ on disk
- buffers: 1 input, $k-m$ output, $p = N-(k-m)-1$ in-memory partitions

When we come to scan and partition *S* relation

- any tuple with hash in range $0..m-1$ can be resolved
- other tuples are written to one of k partition files for *S*

Final phase is same as grace join, but with only k partitions.

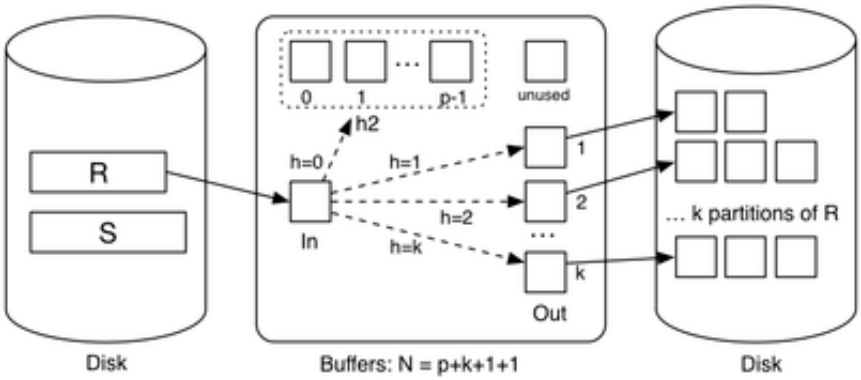
Comparison:

- grace hash join creates $N-1$ partitions on disk
- hybrid hash join creates m (memory) + k (disk) partitions

... Hybrid Hash Join

17/69

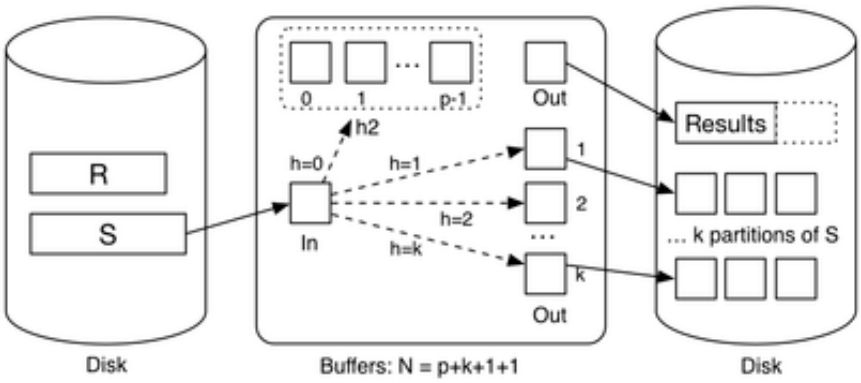
First phase of hybrid hash join with $m=1$ (partitioning *R*):



... Hybrid Hash Join

18/69

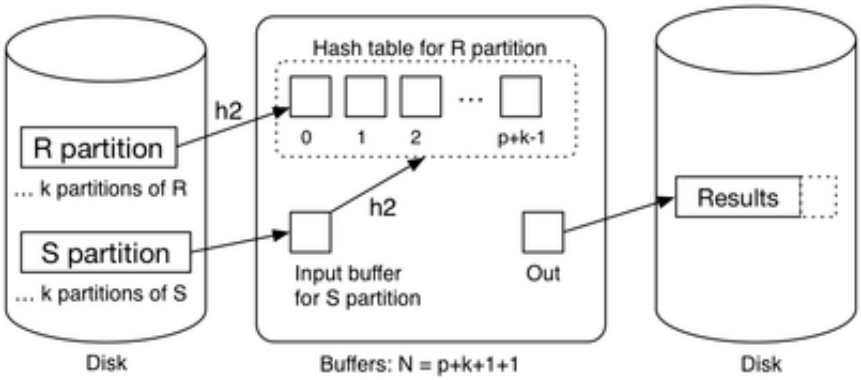
Next phase of hybrid hash join with $m=1$ (partitioning *S*):



... Hybrid Hash Join

19/69

Final phase of hybrid hash join with *m=1* (finishing join):



... Hybrid Hash Join

20/69

Some observations:

- with k partitions, each partition has expected size b_R/k
- holding m partitions in memory needs $\lceil mb_R/k \rceil$ buffers
- with k-m output buffers, must have $\lceil mb_R/k \rceil + (k-m) + 1 = N$

Other notes:

- if $N = b_R+2$, using block nested loop join is simpler
- cost depends on N (but less than grace hash join)

Best-cost scenario:

- $m = 1$, $k \cong \lceil b_R/N \rceil$ (satisfying above constraint)

Exercise 4: Hybrid Hash Join Cost

21/69

Consider executing *Join[i=j](R,S)* with the following parameters:

- $r_R = 1000$, $b_R = 50$, $r_S = 3000$, $b_S = 150$, $c_{Res} = 30$

- *R.i* is primary key, each *R* tuple joins with 2 *S* tuples
- DBMS has *N = 42* buffers available for the join
- data + hash have reasonably uniform distribution

Calculate the cost for evaluating the above join

- using hybrid hash join with *m=1, p=40*
- compute #pages read/written
- compute #join-condition checks performed
- assume that no *R* partition is larger than 40 pages

Join Summary

22/69

No single join algorithm is superior in some overall sense.

Which algorithm is best for a given query depends on:

- sizes of relations being joined, size of buffer pool
- any indexing on relations, whether relations are sorted
- which attributes and operations are used in the query
- number of tuples in *S* matching each tuple in *R*
- distribution of data values (uniform, skew, ...)

Choosing the "best" join algorithm is critical because the cost difference between best and worst case can be very large.

E.g. *Join[id=stude](Student,Enrolled)*: 3,000 ... 2,000,000

Join in PostgreSQL

23/69

Join implementations are under: **`src/backend/executor`**

PostgreSQL suports three kinds of join:

- nested loop join (**`nodeNestloop.c`**)
- sort-merge join (**`nodeMergejoin.c`**)
- hash join (**`nodeHashjoin.c`**) (hybrid hash join)

Query optimiser chooses appropriate join, by considering

- physical characteristics of tables being joined
- estimated selectivity (likely number of result tuples)

Exercise 5: Outer Join?

24/69

Above discussion was all in terms of theta inner-join.

How would the algorithms above adapt to outer join?

Consider the following ...

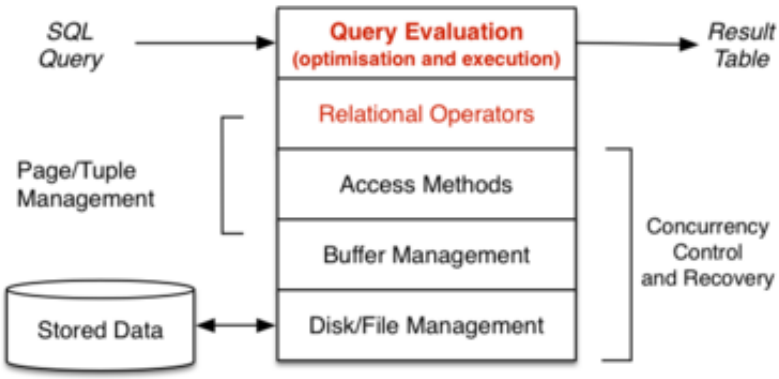
```
select *
from   R left outer join S on (R.i = S.j)

select *
from   R right outer join S on (R.i = S.j)
```

```
select *
from   R full outer join S on (R.i = S.j)
```

Query Processing/Evaluation

Query Evaluation

26/69



... Query Evaluation

27/69

A *query* in SQL:

- states *what* answers are required (declarative)
- does not say *how* they should be computed (procedural)

A *query evaluator/processor* :

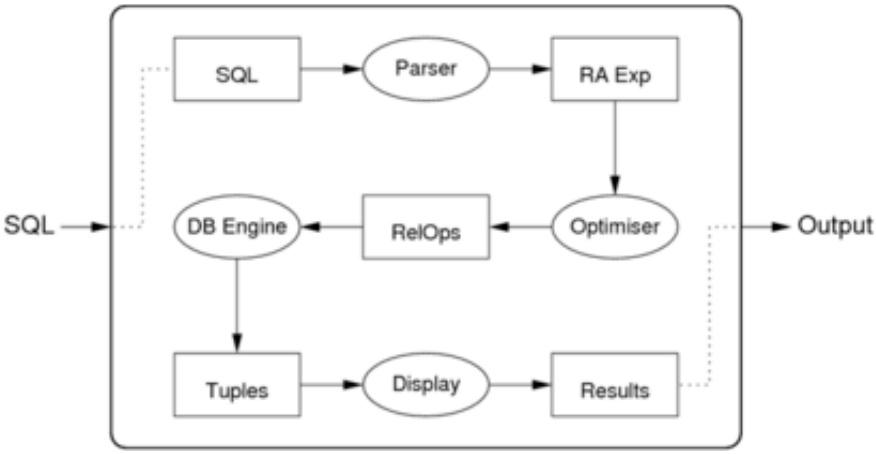
- takes declarative description of query (in SQL)
- parses query to internal representation (relational algebra)
- determines plan for answering query (expressed as DBMS ops)
- executes method via DBMS engine (to produce result tuples)

Some DBMSs can save query plans for later re-use.

... Query Evaluation

28/69

Internals of the query evaluation "black-box":



... Query Evaluation

29/69

DBMSs provide several "flavours" of each RA operation.

For example:

- several "versions" of selection (σ) are available
- each version is effective for a particular kind of selection, e.g

```
select * from R where id = 100  -- hashing
select * from S                 -- Btree index
where age > 18 and age < 35
select * from T                 -- MALH file
where a = 1 and b = 'a' and c = 1.4
```

Similarly, π and $\bowtie$ have versions to match specific query types.

... Query Evaluation

30/69

We call these specialised version of RA operations *RelOps*.

One major task of the query processor:

- given a set of RA operations to be executed
- find a combination of RelOps to do this efficiently

Requires the query translator/optimiser to consider

- information about relations (e.g. sizes, primary keys, ...)
- information about operations (e.g. selection reduces size)

RelOps are realised at execution time

- as a collection of inter-communicating *nodes*
- communicating either via pipelines or temporary relations

Terminology Variations

31/69

Relational algebra expression of SQL query

- intermediate query representation
- logical query plan

Execution plan as collection of RelOps

- query evaluation plan
- query execution plan
- physical query plan

Representation of RA operators and expressions

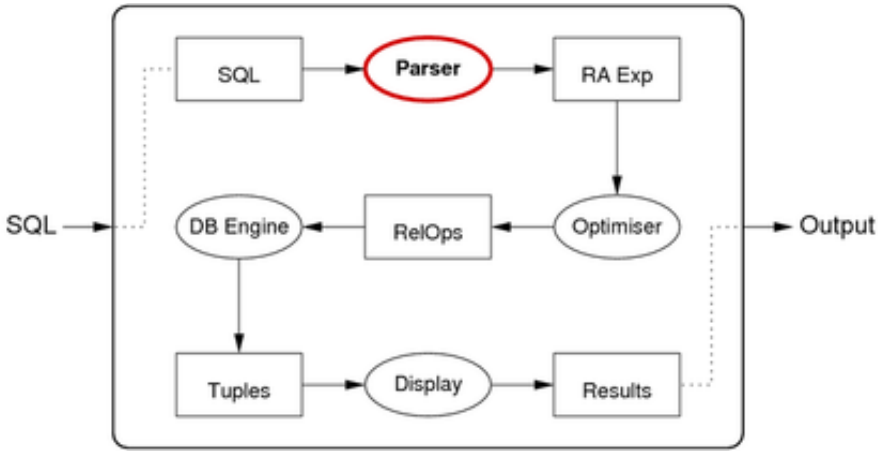
- σ = *Select* = *Sel*, π = *Project* = *Proj*
- $R \bowtie S$ = *R Join S* = *Join(R,S)*, $\wedge$ = &, $\vee$ = |

Query Translation

... Terminology Variations

33/69

Query translation: SQL statement text → RA expression



Query Translation

34/69

Translation step: SQL text → RA expression

Example:

```
SQL: select name from Students where id=7654321;
-- is translated to
RA:  Proj[name](Sel[id=7654321]Students)
```

Processes: lexer/parser, mapping rules, rewriting rules.

Mapping from SQL to RA may include some optimisations, e.g.

```
select * from Students where id = 54321 and age > 50;
-- is translated to
Sel[age>50](Sel[id=54321]Students)
-- rather than ... because of index on id
Sel[id=54321&age>50](Students)
```

Parsing SQL

35/69

Parsing task is similar to that for programming languages.

Language elements:

- keywords: `create`, `select`, `from`, `where`, ...
- identifiers: `Students`, `name`, `id`, `CourseCode`, ...
- operators: `+`, `-`, `=`, `<`, `>`, `AND`, `OR`, `NOT`, `IN`, ...
- constants: `'abc'`, `123`, `3.1`, `'01-jan-1970'`, ...

PostgreSQL parser ...

- implemented via lex/yacc (**`src/backend/parser`**)
- maps all identifiers to lower-case (A-Z → a-z)
- needs to handle user-extendable operator set
- makes extensive use of catalog (**`src/backend/catalog`**)

Mapping SQL to Relational Algebra

A given SQL query typically has many translations to RA.

For example:

```
SELECT s.name, e.subj
FROM   Students s, Enrolments e
WHERE  s.id = e.sid AND e.mark > 50;
```

is equivalent to any of

- $\pi_{s.name,e.subj}(\sigma_{s.id=e.sid \wedge e.mark>50} (Students \times Enrolments))$
- $\pi_{s.name,e.subj}(\sigma_{s.id=e.sid} (\sigma_{e.mark>50} (Students \times Enrolments)))$
- $\pi_{s.name,e.subj}(\sigma_{e.mark>50} (Students \bowtie_{s.id=e.sid} Enrolments)))$

... Mapping SQL to Relational Algebra

More complex example:

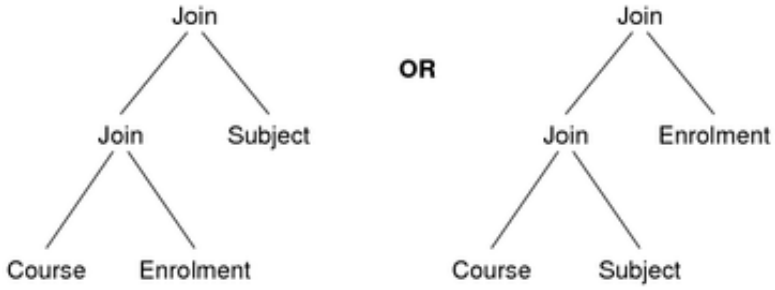
```
select   distinct s.code
from     Course c, Subject s, Enrolment e
where    c.id = e.course and c.subject = s.id
group by s.id  having count(*) > 100;
```

can be translated to the relational algebra expression

$$
\begin{aligned}
&Uniq(Proj_{[code]}(\\
&\quad GroupSelect_{[groupSize>100]}(\\
&\qquad GroupBy_{[s.id]} (\\
&\qquad\quad Enrolment \bowtie Course \bowtie Subjects \\
&))))
\end{aligned}
$$

... Mapping SQL to Relational Algebra

The join operations could be done in two different ways:



Note: for a join on *n* tables, there are potentially *O(n!)* possible trees

The *query optimiser* aims to find version with lowest total cost.

Mapping Rules

39/69

Mapping from SQL → RA expression requires:

- a collection of *templates*, ≥1 for each kind of query
- a process to match an SQL statement to a template
- mapping rules for translating matched query into RA

May need to apply >1 templates to map whole SQL statement.

After mapping, apply rewriting rules to "improve" RA expression

- convert to equivalent, simpler, more efficient expression

Note: PostgreSQL also has user-defined mapping rules (`CREATE RULE`)

... Mapping Rules

40/69

Projection:

`SELECT` *a+b* `AS` *x, c* `AS` *y* `FROM` *R ...*

⇒ $Proj_{[x \leftarrow a+b,\ y \leftarrow c]}(R)$

SQL projection extends RA projection with renaming and assignment

Join:

`SELECT` *...* `FROM` *... R, S ...* `WHERE` *... R.f op S.g ...* , or

`SELECT` *...* `FROM` *... R* `JOIN` *S* `ON` *(R.f op S.g) ...* `WHERE` *...*

⇒ $Join_{[R.f\ op\ S.g\ ]}(R,S)$

... Mapping Rules

41/69

Selection:

`SELECT` *...* `FROM` *... R ...* `WHERE` *... R.f op val ...*

⇒ $Select_{[R.f\ op\ val]}(R)$

SELECT ... FROM ... R ... WHERE ... $Cond_{1,R}$ AND $Cond_{2,R}$ ...

⇒ $Select_{[Cond_{1,R}\ \&\ Cond_{2,R}]}(R)$
or
⇒ $Select_{[Cond_{1,R}]}(Select_{[Cond_{2,R}]}(R))$

Exercise 6: Mapping OR expressions

42/69

Possible mappings for `WHERE` expressions with `AND` are

SELECT ... FROM ... R ... WHERE ... X AND Y ...

⇒ $Select_{[X\ \&\ Y]}(R)$ or $Select_{[X]}(Select_{[Y]}(R))$

What are possible mappings for

SELECT ... FROM ... R ... WHERE ... X OR Y ...

Use these to translate:

```
select * from R where (a=1 or a=3) and b < c
```

Mapping Rules

43/69

Aggregation operators (e.g. `MAX`, `SUM`, ...):

- add as new operators in extended RA
  e.g. `SELECT MAX(age) FROM ...` ⇒ $max(Proj_{[age]}(...))$

Sorting (`ORDER BY`):

- add *Sort* operator into extended RA (e.g. $Sort_{[+name,-age]}(...)$)

Duplicate elimination (`DISTINCT`):

- add *Uniq* operator into extended RA (e.g. *Uniq(Proj(...))*)

Grouping (`GROUP BY`, `HAVING`):

- add operators into extended RA (e.g. *GroupBy, GroupSelect*)

... Mapping Rules

44/69

View example: assuming *Employee(id,name,birthdate,salary)*

```
-- view definition
create view OldEmps as
select * from Employees
where  birthdate < '01-01-1960';
-- view usage
select name from OldEmps;
```

yields

- *OldEmps = $Select_{[birthdate<'01-01-1960']}$(Employees)*
- $Proj_{name}$*(OldEmps)*

⇒ $Proj_{name}(Select_{[birthdate<'01-01-1960']}(Employees))$

Exercise 7: Mapping Views

Given the following definitions:

```
create table R(a integer, b integer, c integer);

create view RR(f,g,h) as
select * from R where a > 5 and b = c;
```

Show how the following might be mapped to RA:

```
select * from RR where f > 10;
```

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):

- $\sigma_{c \wedge d}(R) \leftrightarrow \sigma_c(\sigma_d(R))$
- $\sigma_{c \vee d}(R) \leftrightarrow \sigma_c(R) \cup \sigma_d(R)$ (but only if R is a set)

... 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 c refers to 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:

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

Projections can be pushed into joins:

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

where

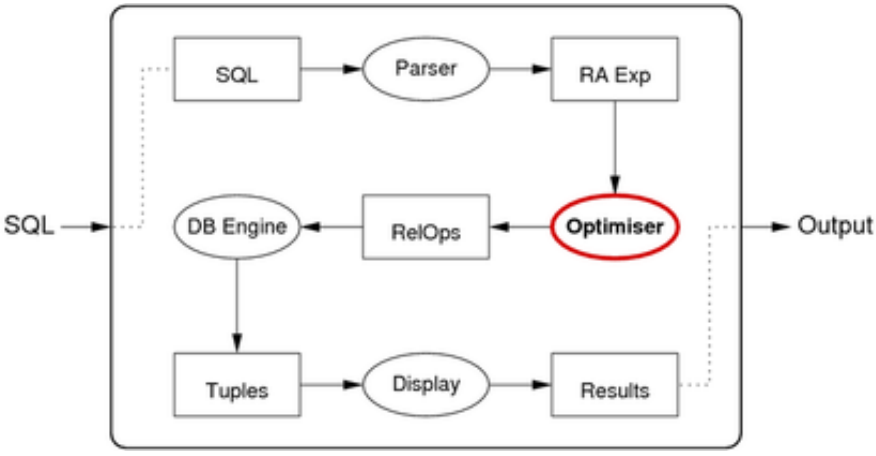
- M and N must contain all attributes needed for c
- M and N must contain all attributes used in L ($L \subset M \cup N$)

Subqueries ⇒ convert to a join

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* provides an efficient evaluation

"Optimisation" is a misnomer since query optimisers

- aim to find a good plan (maybe not optimal)
- within a reasonable amount of time

Observed Time = Planning time + Execution 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)

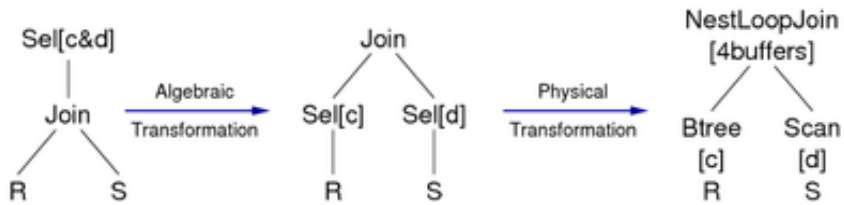
All driven by aim of minimising (or at least reducing) "cost".

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 {
    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 8: Alternative Join Plans

57/69

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 join trees/orders for this query.

Cost Models and Analysis

58/69

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)

59/69

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:

- call to DBMS operation to implement this RA operation

  (may require multiple DBMS operations for a single 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

64/69

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 pages of 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

65/69

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

66/69

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

Estimating size of result for e.g.

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

Could estimate by using:

- uniform distribution assumption, r, #courses

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

... 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.

Exercise 9: 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 $r=1000$

Produced: 9 May 2016