Week 08 Lectures

Implementing Join

Join

2/91

DBMSs are engines to *store*, *combine* and *filter* information.

Join (⋈) is the primary means of *combining* information.

Join is important and potentially expensive

Most common join condition: equijoin, e.g. `(R.pk = S.fk)`

Join varieties (natural, inner, outer, semi, anti) all behave similarly.

We consider three strategies for implementing join

- *nested loop* ... simple, widely applicable, inefficient without buffering
- *sort-merge* ... works best if tables are sorted on join attributes
- *hash-based* ... requires good hash function and sufficient buffering

Join Example

3/91

Consider a university database with the schema:

```
create table Student(
   id     integer primary key,
   name   text,  ...
);
create table Enrolled(
   stude  integer references Student(id),
   subj   text references Subject(code),  ...
);
create table Subject(
   code   text primary key,
   title  text,  ...
);
```

... Join Example

4/91

List names of students in all subjects, arranged by subject.

SQL query to provide this information:

```
select E.subj, S.name
from   Student S, Enrolled E
where  S.id = E.stude
order  by E.subj, S.name;
```

And its relational algebra equivalent:

> *Sort[subj] (Project[subj,name] (Join[id=stude](Student,Enrolled)))*

To simplify formulae, we denote `Student` by *S* and `Enrolled` by *E*

... Join Example

Some database statistics:

Sym	Meaning	Value
r_S	# student records	20,000
r_E	# enrollment records	80,000
c_S	`Student` records/page	20
c_E	`Enrolled` records/page	40
b_S	# data pages in `Student`	1,000
b_E	# data pages in `Enrolled`	2,000

Also, in cost analyses below, N = number of memory buffers.

... Join Example

`Out` = *Student* ⋈ *Enrolled* relation statistics:

Sym	Meaning	Value
r_{Out}	# tuples in result	80,000
C_{Out}	result records/page	80
b_{Out}	# data pages in result	1,000

Notes:

- r_{Out} ... one result tuple for each `Enrolled` tuple
- C_{Out} ... result tuples have only `subj` and `name`
- in analyses, ignore cost of writing result ... same in all methods

---

Nested Loop Join

Basic strategy (R.a ⋈ S.b):

```
Result = {}
for each page i in R {
   pageR = getPage(R,i)
   for each page j in S {
      pageS = getPage(S,j)
      for each pair of tuples tR,tS
                       from pageR,pageS {
         if (tR.a == tS.b)
            Result = Result ∪ (tR:tS)
} } }
```

Needs input buffers for R and S, output buffer for "joined" tuples

Terminology: R is outer relation, S is inner relation

Cost = $b_R \cdot b_S$ ... ouch!

Block Nested Loop Join

Method (for *N* memory buffers):

- read *N–2*–page chunk of *R* into memory buffers
- for each *S* page
  check join condition on all (t_R, t_S) pairs in buffers
- repeat for all *N–2*–page chunks of *R*

... Block Nested Loop Join

Best–case scenario: $b_R \leq N-2$

- read b_R pages of relation *R* into buffers
- while whole *R* is buffered, read b_S pages of *S*

Cost = $b_R + b_S$

Typical–case scenario: $b_R > N-2$

- read $ceil(b_R/(N-2))$ chunks of pages from *R*
- for each chunk, read b_S pages of *S*

Cost = $b_R + b_S \,.\, ceil(b_R/N-2)$

Note: always requires $r_R.r_S$ checks of the join condition

Exercise 1: Nested Loop Join Cost

Compute the cost (# pages fetched) of $(S \bowtie E)$

Sym	Meaning	Value
r_S	# student records	20,000
r_E	# enrollment records	80,000
c_S	`Student` records/page	20
c_E	`Enrolled` records/page	40
b_S	# data pages in `Student`	1,000
b_E	# data pages in `Enrolled`	2,000

for *N = 22, 202, 2002* and different inner/outer combinations

---

If the query in the above example was:

```
select j.code, j.title, s.name
from   Student s
       join Enrolled e on (s.id=e.student)
       join Subject j on (e.subj=j.code)
```

how would this change the previous analysis?

What join combinations are there?

Assume 2000 subjects, with $c_J = 10$

How large would the intermediate tuples be? What assumptions?

Compute the cost (# pages fetched, # pages written) for N = 202

... Block Nested Loop Join

Why block nested loop join is actually useful in practice ...

Many queries have the form

```
select * from R,S where r.i=s.j and r.x=K
```

This would typically be evaluated as

```
Tmp = Sel[x=K](R)
Res = Join[i=j](Tmp, S)
```

If `Tmp` is small ⇒ may fit in memory (in small #buffers)

Index Nested Loop Join

A problem with nested-loop join:

- needs repeated scans of *entire* inner relation *S*

If there is an index on *S*, we can avoid such repeated scanning.

Consider *Join[i=j](R,S)*:

```
for each tuple r in relation R {
    use index to select tuples
        from S where s.j = r.i
    for each selected tuple s from S {
        add (r,s) to result
}   }
```

... Index Nested Loop Join

This method requires:

- one scan of *R* relation (b_R)
  - only one buffer needed, since we use *R* tuple-at-a-time
- for each *tuple* in *R* (r_R), one index lookup on *S*
  - cost depends on type of index and number of results
  - best case is when each *R.i* matches few *S* tuples

Cost = $b_R + r_R . Sel_S$ (Sel_S is the cost of performing a select on *S*).

Typical Sel_S = 1–2 (hashing) .. b_q (unclustered index)

Trade-off: $r_R . Sel_S$ vs $b_R . b_S$, where $b_R \ll r_R$ and $Sel_S \ll b_S$

Exercise 2: Index Nested Loop Join Cost

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

- $r_S = 1000$, $b_S = 50$, $r_T = 3000$, $b_T = 600$
- *S.i* is primary key, and *T* has index on *T.j*
- *T* is sorted on *T.j*, each *S* tuple joins with 2 *T* tuples
- DBMS has *N = 12* buffers available for the join

Calculate the costs for evaluating the above join

- using block nested loop join
- using index nested loop join

$Cost_r$ = # pages read and $Cost_j$ = # join-condition checks

---

Sort-Merge Join

Basic approach:

- sort both relations on join attribute (reminder: *Join [i=j] (R,S)*)
- scan together using *merge* to form result `(r,s)` tuples

Advantages:

- no need to deal with "entire" *S* relation for each *r* tuple
- deal with runs of matching *R* and *S* tuples

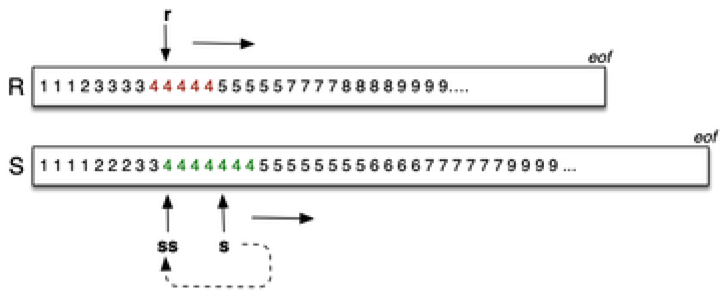
Disadvantages:

- cost of sorting both relations (already sorted on join key?)
- some rescanning required when long runs of *S* tuples

... Sort-Merge Join

Method requires several cursors to scan sorted relations:

- `r` = current record in *R* relation
- `s` = start of current run in *S* relation
- `ss` = current record in current run in *S* relation



... Sort-Merge Join

Algorithm using query iterators/scanners:

```
Query ri, si;  Tuple r,s;

ri = startScan("SortedR");
si = startScan("SortedS");
while ((r = nextTuple(ri)) != NULL
       && (s = nextTuple(si)) != NULL) {
```

```
    // align cursors to start of next common run
    while (r != NULL && r.i < s.j)
           r = nextTuple(ri);
    if (r == NULL) break;
    while (s != NULL && r.i > s.j)
           s = nextTuple(si);
    if (s == NULL) break;
    // must have (r.i == s.j) here
...
```

... Sort-Merge Join


```
...
    // remember start of current run in S
    TupleID startRun = scanCurrent(si)
    // scan common run, generating result tuples
    while (r != NULL && r.i == s.j) {
        while (s != NULL and s.j == r.i) {
            addTuple(outbuf, combine(r,s));
            if (isFull(outbuf)) {
                writePage(outf, outp++, outbuf);
                clearBuf(outbuf);
            }
            s = nextTuple(si);
        }
        r = nextTuple(ri);
        setScan(si, startRun);
    }
}
```

... Sort-Merge Join

Buffer requirements:

- for sort phase:
  - as many as possible (remembering that cost is $O(log_N)$)
  - if insufficient buffers, sorting cost can dominate
- for merge phase:
  - one output buffer for result
  - one input buffer for relation *R*
  - (preferably) enough buffers for longest run in *S*

... Sort-Merge Join

Cost of sort-merge join.

Step 1: sort each relation (if not already sorted):

- Cost = $2.b_R\ (1 + log_{N-1}(b_R\ /N))\ +\ 2.b_S\ (1 + log_{N-1}(b_S\ /N))$
  (where N = number of memory buffers)

Step 2: merge sorted relations:

- if every run of values in *S* fits completely in buffers,
  merge requires single scan, Cost = $b_R + b_S$
- if some runs in of values in *S* are larger than buffers,
  need to re-scan run for each corresponding value from *R*

Sort-Merge Join on Example

Case 1: *Join[id=stude](Student,Enrolled)*

- relations are not sorted on *id#*
- memory buffers $N=32$; all runs are of length < 30

$$\begin{aligned} Cost &= sort(S) + sort(E) + b_S + b_E \\ &= 2b_S(1+\log_{31}(b_S/32)) + 2b_E(1+\log_{31}(b_E/32)) + b_S + b_E \\ &= 2\times1000\times(1+2) + 2\times2000\times(1+2) + 1000 + 2000 \\ &= 6000 + 12000 + 1000 + 2000 \\ &= 21{,}000 \end{aligned}$$

... Sort-Merge Join on Example

23/91

Case 2: *Join[id=stude](Student,Enrolled)*

- *Student* and *Enrolled* already sorted on *id#*
- memory buffers $N=4$ (S input, 2 × E input, output)
- 5% of the "runs" in E span two pages
- there are no "runs" in S, since *id#* is a primary key

For the above, no re-scans of E runs are ever needed

$Cost = 2{,}000 + 1{,}000 = 3{,}000$ (regardless of which relation is outer)

Exercise 3: Sort-merge Join Cost

24/91

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

- $r_S = 1000$, $b_S = 50$, $r_T = 3000$, $b_T = 150$
- $S.i$ is primary key, and T has index on $T.j$
- T is sorted on $T.j$, each S tuple joins with 2 T tuples
- DBMS has $N = 42$ buffers available for the join

Calculate the cost for evaluating the above join

- using sort-merge join
- compute #pages read/written
- compute #join-condition checks performed

Hash Join

25/91

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:

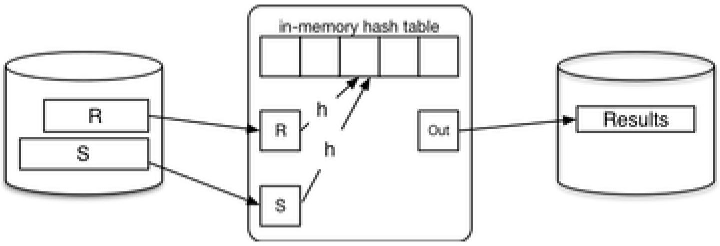
- hash part of outer relation *R* into memory buffers (build)
- scan inner relation *S*, using hash to search (probe)
  - if R.i=S.j, then h(R.i)=h(S.j) (hash to same buffer)
  - only need to check one memory buffer for each *S* tuple
- repeat until whole of *R* has been processed

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)]
}
```

Best case: # join tests ≤ $r_S.c_R$ (cf. nested-loop $r_S.r_R$)

... Simple Hash Join

Cost for simple hash join ...

Best case: all tuples of R fit in the hash table

- Cost = b_R + b_R Bs
- Same page reads as block nested loop, but less join tests

Good case: refill hash table *m* times (where $m \geqslant ceil(b_R \,/\, (N-3))$)

- Cost = b_R + $m.b_R$ Bs
- More page reads that block nested loop, but less join tests

Worst case: everything hashes to same page

- Cost = $b_R + b_R.b_S$

Exercise 4: Simple Hash Join Cost

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 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

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 $\geqslant \sqrt{b_R}$ 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

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

... Grace Hash Join

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

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 5: Grace Hash Join Cost

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

Exercise 6: Grace Hash Join Cost

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

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* for 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.

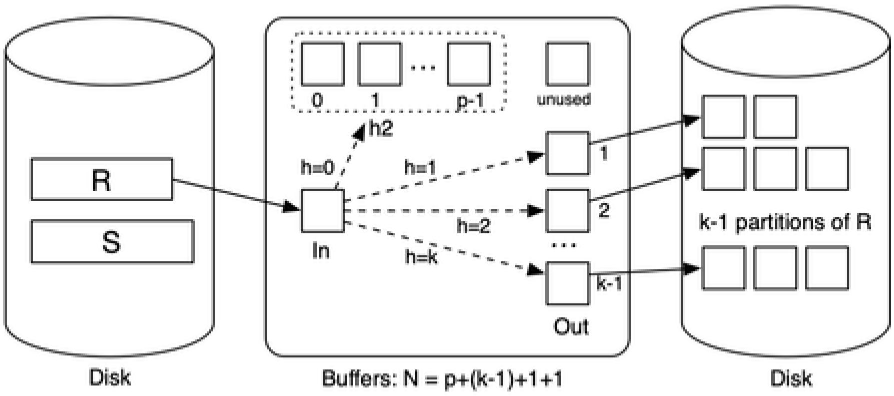
Comparison:

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

---

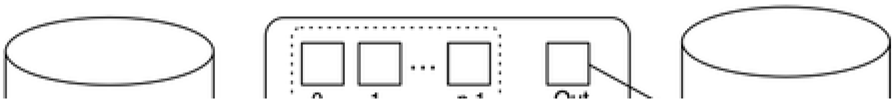
... Hybrid Hash Join

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



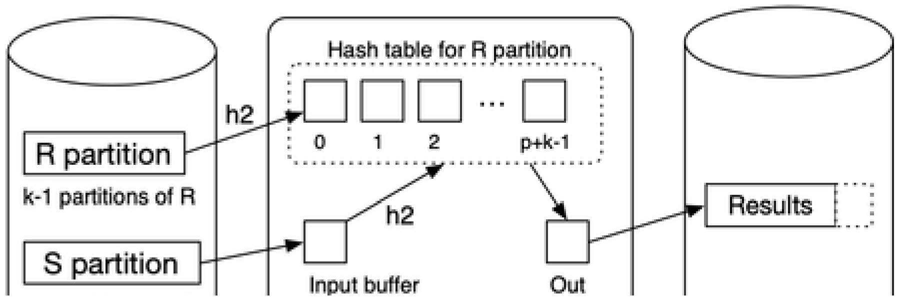
... Hybrid Hash Join

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



... Hybrid Hash Join

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



... Hybrid Hash Join

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
- trade-off between in-memory partition space and #partitions

Best-cost scenario:

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

Other notes:

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

Exercise 7: Hybrid Hash Join Cost

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

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

44/91

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 8: Outer Join?

45/91

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 Evaluation

Query Evaluation

47/91

... Query Evaluation

48/91

A *query* in SQL:

- states *what* kind of 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

Internals of the query evaluation "black-box":

... Query Evaluation

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

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

One major task of the query processor:

- given a RA expression to be evaluated
- 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

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 ⋈ S = R Join S = Join(R,S)*, *∧ = &*, *∨ = |*

Query Translation

Query translation: SQL statement text → RA expression

Query Translation

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

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`**)

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

57/91

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

58/91

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:

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

---

... Relational Algebra Laws

59/91

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 ($L \subset M \cup N$)

Query Rewriting

60/91

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
```

... Query Rewriting

But not all subqueries can be converted to join, e.g.

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

has to be evaluated as

Val = max[id]Students

$Res = \pi_{(sid,cid)}(\sigma_{sid=Val}Enrolments)$

... Query Rewriting

In PostgreSQL, views are implemented via rewrite rules

- a reference to view in SQL expands to its definition in RA

Example:

```
create view COMP9315studes as
select stu,mark from Enrolments where course='COMP9315';
-- students who passed
select stu from COMP9315studes where mark >= 50;
```

is represented in RA by

```
COMP9315studes
  = Proj[stu,mark](Sel[course=COMP9315](Enrolments))
-- with query ...
Proj[stu](Sel[mark>=50](COMP9315studes))
-- becomes ...
Proj[stu](Sel[mark>=50](
  Proj[stu,mark](Sel[course=COMP9315](Enrolments))))
-- which could be rewritten as ...
Proj[stu](Sel[mark>=50 & course=COMP9315]Enrolments)
```

Exercise 9: SQL → RelAlg

Convert the following queries into (efficient?) RA expressions

```
select * from R where a > 5;

select * from R where id = 1234 and a > 5;
```

```
select R.a from R, S where R.i = S.j;

select R.a from R join S on R.i = S.j;

select * from R, S where R.i = S.j and R.a = 6

select R.a from R, S, T where R.i = S.j and S.k = T.y;
```

Assume `R.id` is a primary key and `R` is hashed on `id`

Assume that there is a B-tree index on `R.b`

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;  cost = 0
      for each node e of RA' (recursively) {
         ROp = select RelOp method for e
         Plan = Plan ∪ ROp
         cost += Cost(ROp) // using child info
      }
      if (cost < MinCost)
         { MinCost = cost;  BestPlan = Plan }
   }
}
```

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

Exercise 10: 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='19T2'
       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` $\Rightarrow$ `indexSearch[A=c](R)`
- $\sigma_{A=c}(R)$ and `R` is hashed on `A` $\Rightarrow$ `hashSearch[A=c](R)`
- $\sigma_{A=c}(R)$ and `R` is sorted on `A` $\Rightarrow$ `binarySearch[A=c](R)`
- $\sigma_{A \geqslant c}(R)$ and `R` has clustered index on `A`

  $\Rightarrow$ `indexSearch[A=c](R)` then scan
- $\sigma_{A \geqslant c}(R)$ and `R` is hashed on `A`

  $\Rightarrow$ `linearSearch[A>=c](R)`

... Choosing Access Methods (RelOps)

Rules for choosing $\bowtie$ access methods:

- $R \bowtie S$ and `R` fits in memory buffers $\Rightarrow$ `bnlJoin(R,S)`
- $R \bowtie S$ and `S` fits in memory buffers $\Rightarrow$ `bnlJoin(S,R)`
- $R \bowtie S$ and `R,S` sorted on join attr $\Rightarrow$ `smJoin(R,S)`

- $R \bowtie S$ and `R` has index on join attr $\Rightarrow$ `inlJoin(S,R)`
- $R \bowtie S$ and no indexes, no sorting $\Rightarrow$ `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 estimated 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

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

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

... Estimating Selection Result Size

Estimating size of result for e.g.

```
select * from Enrolment where year > 2015;
```

Could estimate by using:

- uniform distribution assumption, r, min/max years

Assume: min(year)=2010, max(year)=2019, $|Enrolment|=10^5$

- 10^5 from 2010-2019 means approx 10000 enrolments/year
- this suggests 40000 enrolments since 2016

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, 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 11: 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

84/91

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 ≤ min(R.a)* ⇒ 0 results

Case 2: *k > max(R.a)* ⇒ ≅ r_R results

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

Estimating Join Result Size

85/91

Analysis relies on semantic knowledge about data/relations.

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

Case 1: *values(R.a) ∩ values(S.a) = {}* ⇒ $size(R \bowtie_a S) = 0$

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

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

Exercise 12: Join Size Estimation

86/91

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

87/91

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

PostgreSQL Query Optimization

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`**

... PostgreSQL Query Optimization

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.

... PostgreSQL Query Optimization

Produced: 1 May 2020