CS411
Database Systems

12: Query Optimization

Kazuhiro Minami

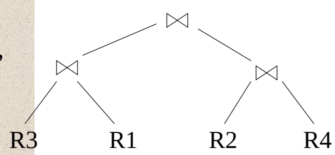
The order that relations are joined in has a huge impact on performance

Given:

query **R1 ⋈ … ⋈ Rn**,

function **cost( )**,

find the best join tree for the query



Plan = *tree*

Partial plan = *subtree*

Dynamic programming is a good (bottom-up) way to choose join ordering

Find the best plan for each subquery Q of {R1, …, Rn}:

1. {R1}, …, {Rn}
2. {R1, R2}, {R1, R3}, …, {Rn-1, Rn}
3. {R1, R2, R3}, {R1, R2, R4}, …
4. …
5. {R1, …, Rn}

Output:

1. A best plan Plan(Q)
2. Cost(Q)
3. Size(Q)

The *i*th step of the dynamic program

For each Q ⊆ {R1, …, Rn} of size *i* do:

1. For every pair Q1, Q2 such that Q = Q1 ∪ Q2, compute cost(Plan(Q1) ⋈ Plan(Q2))
   Cost(Q) = the smallest such cost
   Plan(Q) = the corresponding plan
2. Compute Size(Q)

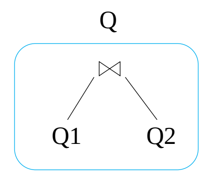
Dynamic Programming

- Return Plan({R1, …, Rn})

Computing the Cost of a Plan Recursively

To illustrate, we will make the following simplifications:

- Cost(P1 ⋈ P2) = Cost(P1) + Cost(P2) + size(intermediate results for P1 and P2)
- Intermediate results:
  - If P1 is a join, then the size of the intermediate result is size(P1), otherwise the size is 0
  - Similarly for P2
- Cost of a scan = 0



Example

- Cost(R5 ⋈ R7)

  = Cost(R5) + Cost(R7)

  + intermediate results for R5 and R7

  = 0 (no intermediate results)
- Cost((R2 ⋈ R1) ⋈ R7)

  = Cost(R2 ⋈ R1) + Cost(R7) + size(R2 ⋈ R1)

  = size(R2 ⋈ R1)

Intermediate result of R2 ⋈ R1

Rough Estimation of a Plan Size

- Relations: R, S, T, U
- Number of tuples: 2000, 5000, 3000, 1000
- Size estimation: T(A ⋈ B) = 0.01*T(A)*T(B)

$R \bowtie S \bowtie T \bowtie U$

Number of tuples:
- R = 2000
- S = 5000
- T = 3000
- U = 1000

Size estimate:
$size(A \bowtie B) = .01*size(A) *size(B)$

Unrealistic!

Subquery	Size	Lowest Cost	Plan
RS			
RT			
RU			
ST			
SU			
TU			
RST			
RSU			
RTU			
STU			
RSTU			

We Actually Start with Subqueris of Size 1

Subquery	Size	Lowest Cost	Plan
R			
S			
T			
U			

$R \bowtie S \bowtie T \bowtie U$

Number of tuples:
- R = 2000
- S = 5000
- T = 3000
- U = 1000

Size estimate:
$size(A \bowtie B) = .01*size(A) *size(B)$

Unrealistic!

Subquery	Size	Lowest Cost	Plan
RS	100k	0	RS
RT	60k	0	RT
RU	20k	0	RU
ST	150k	0	ST
SU	50k	0	SU
TU	30k	0	TU
RST	3M	0 + 0 + 0 + 60k	(RT)S
RSU	1M	20k	(RU)S
RTU	0.6M	20k	(RU)T
STU	1.5M	30k	(TU)S
RSTU	30M	60k+50k= 110k	(RT)(SU)

Join order options for RSTU

- Cost of (RST)U = 60K + 0 + 3M + 0
- Cost of (RSU)T = 20K + 0 + 1M + 0
- Cost of (RTU)S = 20K + 0 + .6M + 0
- Cost of (STU)R = 30K + 0 + 1.5M + 0
- Cost of (RS)(TU) = 0 + 0 + 100K + 30K
- Cost of (RT)(SU) = 0 + 0 + 60K + 50K
- Cost of (RU)(TS) = 0 + 0 + 20K + 150K

What if we don't oversimplify?

- More realistic size/cost estimations!! (next slides)
- Use heuristics to reduce the search space
  - Consider only left linear trees
  - No trees with cartesian products:

R(A,B) S(B,C) T(C,D)

(R ⋈ T) ⋈ S has a cartesian product

Completing a Physical Query Plan

Completing the Physical Query Plan

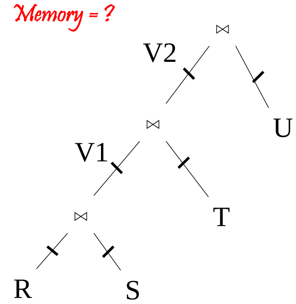
- Choose algorithm to implement each operator

  Need to consider more than I/O cost:
  - How much memory do we have ?
  - Are the input operand(s) sorted ?
- Decide for each intermediate result:
  - Materialize
  - Pipeline

One option is to materialize intermediate results between operators

Cost = ?

Memory = ?



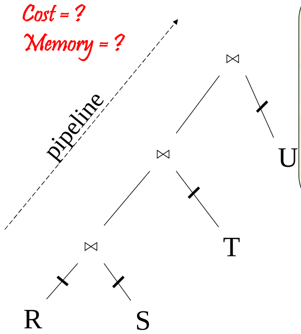

```
HashTable ← S
repeat  read(R, x)
        y ← join(HashTable, x)
        write(V1, y)

HashTable ← T
repeat  read(V1, y)
        z ← join(HashTable, y)
        write(V2, z)

HashTable ← U
repeat  read(V2, z)
        u ← join(HashTable, z)
        write(Answer, u)
```

The second option is to pipeline between operators

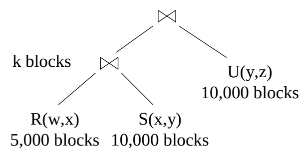
Cost = ?
Memory = ?




```
HashTable1 ← S
HashTable2 ← T
HashTable3 ← U
repeat  read(R, x)
        y ← join(HashTable1, x)
        z ← join(HashTable2, y)
        u ← join(HashTable3, z)
        write(Answer, u)
```

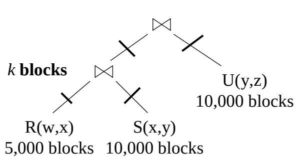
Example 16.36

Logical plan:



Main memory M = 101 blocks of space

Example 16.36



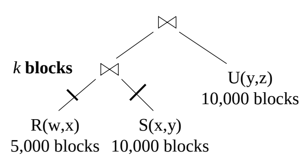
Naive evaluation:

2 partitioned hash-joins, materialized

(Make sure buckets fit in memory!)

Cost 3B(R) + 3B(S) + 4*k* + 3B(U) = 75000 + 4*k*

Example 16.36

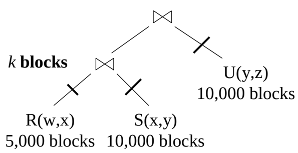


Smarter:

- Step 1: hash R on x into 100 buckets, each of 50 blocks; to disk
- Step 2: hash S on x into 100 buckets; to disk
- Step 3: read each R bucket in memory (50 buffers at a time), join with S (1 buffer at a time); hash result on y into 50 buckets (50 buffers) -- here we *pipeline*

Cost so far: 3B(R) + 3B(S)

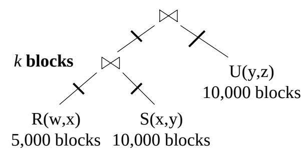
Example 16.36



Continuing:

- How large are the 50 buckets on y? *k*/50 blocks each.
- If *k* <= 50 then keep all 50 buckets in Step 3 in memory, then:
- Step 4: read U from disk, hash on y and join in memory
- Total cost: 3B(R) + 3B(S) + B(U) = 55,000

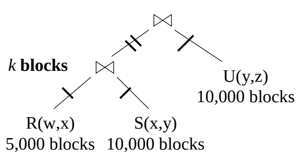
Example 16.36



Continuing:

- If 50 < *k* <= 5000 then send the 50 buckets in Step 3 to disk
  - Each bucket has size *k*/50 <= 100, i.e., it will fit into memory
- Step 4: partition U into 50 buckets
- Step 5: read each partition and join in memory
- Total cost: 3B(R) + 3B(S) + 2*k* + 3B(U) = 75,000 + 2*k*

Example 16.36



Continuing:

- If *k* > 5000, then 50 blocks of memory would make each bucket of the intermediate result too big to fit into memory: materialize, use a second pass to partition the *k* blocks, instead of pipelining them
- 2 partitioned hash-joins
- Cost 3B(R) + 3B(S) + 4*k*+ 3B(U) = 75000 + 4*k*

Example 16.36

Summary:

- If *k* <= 50, cost = 55,000
- If 50 < *k* <=5000, cost = 75,000 + 2*k*
- If *k* > 5000, cost = 75,000 + 4*k*

Estimating Intermediate Result Sizes

because what algorithm you should or could use depends very strongly on the sizes of the relations

Still an area of research today

The number of tuples after projection is:

Easy: $T(\Pi_L(R)) = T(R)$

Because projections don't eliminate duplicates

But the size of each tuple is smaller, of course

The number of tuples after selection is:

$S = \sigma_{A=c}(R)$

- $0 <= T(S) <= T(R)$
- Expected value: $T(S) = T(R)/V(R,A)$

$S = \sigma_{A<c}(R)$

- $T(S)$ can be anything from 0 to $T(R)$
- Heuristic: $T(S) = T(R)/3$

A good guess, though never true in practice. Does not require storing many statistics.

Of course one can do better!

The number of tuples after a join is:

$R \bowtie_A S$

- When the set of A values are disjoint, then $T(R \bowtie_A S) = 0$
- When A is a key in S and a foreign key in R, then $T(R \bowtie_A S) = T(R)$
- When A is a key in both R and S, then $T(R \bowtie_A S) = \min(T(R), T(S))$

Otherwise…

Some assumptions to help us guess the number of tuples resulting from a join:

Containment of values: if $V(R,A) <= V(S,A)$, then the set of A values of R is included in the set of A values of S

(True if A is a foreign key in R, and a key in S)

Preservation of values: for any other attribute B,

$V(R \bowtie_A S, B) = V(R, B)$ (or $V(S, B)$)

The number of tuples after a join is…

If $V(R,A) <= V(S,A)$

Then we expect each tuple t in R to join *some* tuples in S

- How many? The fraction of S that has one particular value.
- On average $T(S)/V(S,A)$
- On average t contributes $T(S)/V(S,A)$ tuples to $R \bowtie_A S$

Hence $T(R \bowtie_A S) = T(R)\, T(S) / V(S,A)$

In general: $T(R \bowtie_A S) = T(R)\, T(S) / \max(V(R,A),V(S,A))$

Example of estimating the number of tuples after a join

$T(R) = 10{,}000$ $T(S) = 20{,}000$

$V(R,A) = 100$ $V(S,A) = 200$

How large is $R \bowtie_A S$?

Answer: $T(R \bowtie_A S) = 10000 * 20000/200 = 1M$

The expected number of tuples after a join on multiple attributes is:

$T(R \bowtie_{A,B} S) =$

$T(R)\, T(S)/[\max(V(R,A),V(S,A))\max(V(R,B),V(S,B))]$

Histograms tell you how many tuples have R.A values within a certain range

- Maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

An example histogram on salary:

Employee(ssn, name, salary, phone)

Salary:	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
Tuples	200	800	5000	12000	6500	500

T(Employee) = 25000, but now we know the distribution

We can use histograms to estimate the size of Employee $\bowtie_{Salary}$ Ranks

Ranks(rankName, salary)

Employee .Salary	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	200	800	5000	12000	6500	500

Ranks. Salary	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	8	20	40	80	100	2

If we don't know how many distinct values there are in each bin, we can estimate:

- V(Employee, Salary) = 200
- V(Ranks, Salary) = 250

Then T(Employee $\bowtie_{Salary}$ Ranks) =

= $\Sigma_{\text{all bins } i}$ T(Emp$_i$) * T(Ranks$_i$)/ 250

= (200*8 + 800*20 + 5000*40 + 12000*80 + 6500*100 + 500*2)/250

= ….

Summary of query optimization process

1. Parse your query into tree form
2. Move selections as far down the tree as you can
3. Project out unwanted attributes as early as you can, when you have their tuples in memory anyway
4. Pick a good join order, based on the **expected size** of intermediate results
5. Pick an **implementation** for each operation in the tree