CSE 562
Database Systems

Query Processing: Physical Operators

Some slides are based or modified from originals by
Database Systems: The Complete Book, Pearson Prentice Hall 2nd Edition
©2008 Garcia-Molina, Ullman, and Widom

cse@buffalo

Outline – Query Optimization

- Overview
- Relational algebra level
  - Algebraic Transformations
- Detailed query plan level
  - Estimate Costs
    - Estimating size of results
    - Estimating # of IOs
  - Generate and compare plans

Algorithms for Algebra Operators

- Three primary techniques
  - Sorting
  - Hashing
  - Indexing
- Three degrees of difficulty
  - data small enough to fit in memory
  - too large to fit in main memory but small enough to be handled by a "two-pass" algorithm
  - so large that "two-pass" methods have to be generalized to "multi-pass" methods (quite unlikely nowadays)

Estimating IOs

- Count # of disk blocks that must be read (or written) to execute query plan

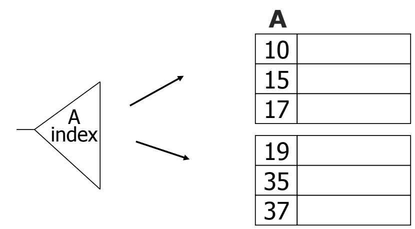
Additional Cost Estimation Parameters

- **B**(R) = # of blocks containing R tuples
- **f**(R) = max # of tuples of R per block
- **M** = # memory blocks available

- **HT**(i) = # levels in index i
- **LB**(i) = # of leaf blocks in index i

Clustering Index

- Index that allows tuples to be read in an order that corresponds to physical order



Clustering Can Radically Change Cost

- Clustered file organization

  R1 R2 S1 S2 | R3 R4 S3 S4 …

- Clustered relation

  R1 R2 R3 R4 | R5 R5 R7 R8 …

- Clustering index

Example

R1 $\bowtie$ **R2** over common attribute **C**

T(R1) = 10,000
T(R2) = 5,000
S(R1) = S(R2) = 1/10 block
Memory available = 101 blocks

→ Metric: # of IOs
(ignoring writing of result)

Caution!

This may not be the best way to compare

- ignoring CPU costs
- ignoring timing
- ignoring double buffering requirements

Options

- Transformations: R1 $\bowtie$ R2, R2 $\bowtie$ R1
- Join algorithms:
  - Iteration (nested loops)
  - Merge join
  - Join with index
  - Hash join

Example

- **Iteration Join** (conceptually – without taking into account disk block issues)

    for each r $\in$ R1 do
        for each s $\in$ R2 do
            if r.C = s.C then output r,s pair

Example

- **Merge Join** (conceptually)

  (1) if R1 and R2 not sorted, sort them
  (2) i $\leftarrow$ 1; j $\leftarrow$ 1;
      While (i $\leq$ T(R1)) $\wedge$ (j $\leq$ T(R2)) do
          if R1{ i }.C = R2{ j }.C then *outputTuples*
          else if R1{ i }.C > R2{ j }.C then j $\leftarrow$ j+1
          else if R1{ i }.C < R2{ j }.C then i $\leftarrow$ i+1

Example

Procedure *outputTuples*

```
While (R1{ i }.C = R2{ j }.C) ∧ (i ≤ T(R1)) do
   [ jj ← j;
     while (R1{ i }.C = R2{ jj }.C) ∧ (jj ≤ T(R2)) do
          [ output pair R1{ i }, R2{ jj };
            jj ← jj+1 ]
     i ← i+1 ]
```


Example

i	R1{i}.C	R2{j}.C	j
1	10	5	1
2	20	20	2
3	20	20	3
4	30	30	4
5	40	30	5
		50	6
		52	7

Example

- **Join with Index** (Conceptually)

```
For each r ∈ R1 do
   [ X  ←  index (R2, C, r.C)
     for each s ∈ X do
          output r,s pair ]
```

Assume R2.C index

Note: X ← index(rel, attr, value)
then X = set of rel tuples with attr = value

Example

- **Hash Join** (Conceptual)
  - Hash function h, range 0 → k
  - Buckets for R1: G0, G1, ... Gk
  - Buckets for R2: H0, H1, ... Hk

Algorithm

1) Hash R1 tuples into G buckets
2) Hash R2 tuples into H buckets
3) For i = 0 to k do
   match tuples in Gi, Hi buckets

Simple Example

hash: even/odd

R1	R2
2	5
4	4
3	12
5	3
8	13
9	8
	11
	14

Buckets

	R1	R2
Even:	2 4 8	4 12 8 14
Odd:	3 5 9	5 3 13 11

Factors that Affect Performance

(1) Tuples of relation stored physically together?

(2) Relations sorted by join attribute?

(3) Indexes exist?

Disk-Oriented Computation Model

- There are *M* main memory buffers
  - Each buffer has the size of a disk block
- The input relation is read one block at a time
- The cost is the number of blocks read
- The output buffers are not part of the *M* buffers mentioned above
  - *Pipelining* allows the output buffers of an operator to be the input of the next one
  - We do not count the cost of writing the output

Notation

- **B(R)** = number of blocks that **R** occupies
- **T(R)** = number of tuples of **R**
- $\mathbf{V(R,[a_1, a_2, \ldots, a_n])}$ = number of distinct tuples in the projection of **R** on $\mathbf{a_1, a_2, \ldots, a_n}$

One-Pass Main Memory Algorithms for Unary Operators

- Assumption: Enough memory to keep the relation
- Projection and selection:
  - Scan the input relation *R* and apply operator one tuple at a time
  - Cost depends on
    - clustering of *R*
    - whether the blocks are consecutive
- Duplicate elimination and aggregation
  - create one entry for each group and compute the aggregated value of the group
  - it becomes hard to assume that CPU cost is negligible
    - main memory data structures are needed

One-Pass Nested Loop Join

- Assume *B(R)* is less than *M*
- Tuples of *R* should be stored in an efficient lookup structure
- **Exercise:** Find the cost of the algorithm below

```
for each block Br of R do
    store tuples of Br in main memory
for each each block Bs of S do
    for each tuple s of Bs
        join tuples of s with matching tuples of R
```


Generalization of Nested-Loops

```
for each chunk of M-1 blocks Br of R do
   store tuples of Br in main memory
   for each block Bs of S do
      for each tuple s of Bs
         join tuples of s with matching tuples of R
```

Exercise: Compute the cost of the above algorithm

Simple Sort-Merge Join

- Assume natural join on *C*
- Sort *R* on *C* using the two-phase multiway merge sort
  - if not already sorted
- Sort *S* on *C*
- Merge (opposite side)
  - assume two pointers `Pr,Ps` to tuples on disk, initially pointing at the start
  - sets `R'`, `S'` in memory
- Remarks:
  - Very low average memory requirement during merging (but no guarantee on how much is needed)
  - **Cost:**

```
while Pr!=EOF and Ps!=EOF
  if *Pr[C] == *Ps[C]
    do_cart_prod(Pr,Ps)
  else if *Pr[C] > *Ps[C]
    Ps++
  else if *Ps[C] > *Pr[C]
    Pr++

function do_cart_prod(Pr,Ps)
  val = *Pr[C]
  while *Pr[C] == val
    store tuple *Pr in set R'
  while *Ps[C] == val
    store tuple *Ps in set S'
  output R' x S' // product
```

Efficient Sort-Merge Join

- **Idea:** Save two disk I/O's per block by combining the second pass of sorting with the "merge"
- **Step 1:**
  Create sorted sublists of size *M* for *R* and *S*
- **Step 2:**
  Bring the first block of each sublist to a buffer
  - assume no more than *M* sublists in all
- **Step 3:**
  Repeatedly find the least *C* value *c* among the first tuples of each sublist. Identify all tuples with join value *c* and join them.
  - When a buffer has no more tuple that has not already been considered load another block into this buffer

Efficient Sort-Merge Join Example

R

C	*RA*
1	r_1
2	r_2
3	r_3
⋮	
20	r_{20}

S

C	*SA*
1	s_1
⋮	
5	s_5
16	s_{16}
⋮	
20	s_{20}

- Assume that after first phase of multiway sort we get 4 sublists, 2 for R and 2 for S
- Also assume that each block contains two tuples

R

Sublist 1	3 7	8 10	11 13	14 16	17 18
Sublist 2	1 2	4 5	6 9	12 15	19 20

S

Sublist 1	1 3	5 17	
Sublist 2	2 4	16 18	19 20

Two-Pass Hash-Based Algorithms

- General Idea: Hash the tuples of the input arguments in such a way that all tuples that must be considered together will have hashed to the same hash value
  - If there are *M* buffers pick *M-1* as the number of hash buckets
- Example: Duplicate Elimination
  - Phase 1: Hash each tuple of each input block into one of the *M-1* bucket/buffers. When a buffer fills, save to disk
  - Phase 2: For each bucket:
    - load the bucket in main memory
    - treat the bucket as a small relation and eliminate duplicates
    - save the bucket back to disk
  - **Catch:** Each bucket has to be less than *M*
  - **Cost:**

Hash-Join Algorithms

- Assuming natural join, use a hash function that
  - is the same for both input arguments *R* and *S*
  - uses only the join attributes
- **Phase 1:** Hash each tuple of *R* into one of the *M-1* buckets R_i and similar each tuple of *S* into one of S_i
- **Phase 2:** For $i=1 \ldots M-1$
  load R_i and S_i in memory
  join them and save result to disk
- **Question:** What is the maximum size of buckets?
- **Question:** Does hashing maintain sorting?

Index-Based Join: Simplest Version

- Assume that we do natural join of R(A,B) and S(B,C) and there is an index on S

```
for each Br in R do
   for each tuple r of Br with B value b
      use index of S to find
         tuples {s1 ,s2 ,...,sn} of S with B=b
      output {rs1 ,rs2 ,...,rsn}
```

- **Cost:** Assuming R is clustered and non-sorted and the index on S is clustered on B then B(R)+T(R)B(S)/V(S,B) + some for reading index

Opportunities in Joins Using Sorted Indexes

- Do a conventional Sort-Join avoiding the sorting of one or both of the input operands

Example 1(a)

Iteration Join R1 $\bowtie$ R2

- Relations <u>not</u> contiguous
- Recall $\left\{\begin{array}{l} T(R1) = 10{,}000 \\ T(R2) = 5{,}000 \\ S(R1) = S(R2) = 1/10 \text{ block} \\ MEM = 101 \text{ blocks} \end{array}\right.$

<u>Cost:</u> for each R1 tuple:

[Read tuple + Read R2]

Total =10,000 [1+5000]=50,010,000 IOs

Can we do better?

<u>Use our memory</u>

(1) Read 100 blocks of R1

(2) Read all of R2 (using 1 block) + join

(3) Repeat until done

Cost

- for each R1 chunk:
  - Read chunk: 1000 IOs
  - Read R2 5000 IOs
  - 6000

Total $= \frac{10{,}000}{1{,}000} \times 6000 = 60{,}000$ IOs

Can we do better?

- **Reverse Join Order: R2 $\bowtie$ R1**

Total $= \frac{5000}{1000} \times (1000 + 10{,}000) =$

$5 \times 11{,}000 = 55{,}000$ IOs

Example 1(b)

Iteration Join R2 $\bowtie$ R1

- Relations contiguous

Cost

For each R2 chunk:

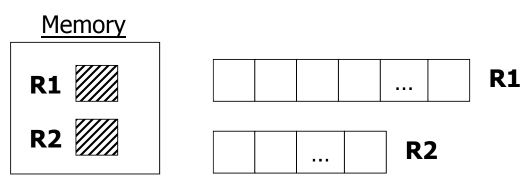
- Read chunk: 100 IOs
- Read R1: 1000 IOs
- 1,100

Total = 5 chunks x 1,100 = 5,500 IOs

Example 1(c)

Merge Join

- Both R1, R2 ordered by C; relations contiguous



Total cost: Read R1 cost + read R2 cost
= 1000 + 500 = 1,500 IOs

Example 1(d)

Merge Join

- R1, R2 <u>not</u> ordered, but contiguous

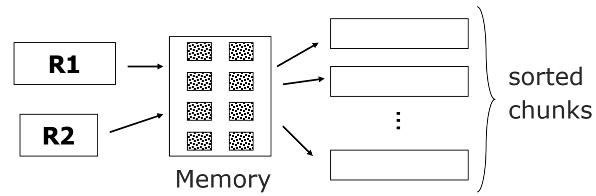
→ Need to sort R1, R2 first… <u>HOW?</u>

One Way to Sort

Merge Sort

(i) For each 100 block chunk of R:

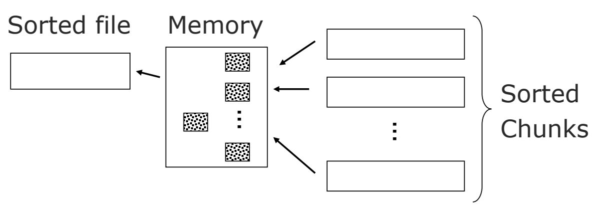
- Read chunk
- Sort in memory
- Write to disk



One Way to Sort

Merge Sort

(ii) Read all chunks + merge + write out



One Way to Sort

<u>Cost:</u> Sort

Each tuple is read, written,
read, written

so...

Sort cost R1: 4 x 1,000 = 4,000

Sort cost R2: 4 x 500 = 2,000

Example 1(d) (Cont.)

Merge Join

- R1,R2 contiguous, but unordered

Total cost = sort cost + join cost
= 6,000 + 1,500 = 7,500 IOs

<u>But:</u> Iteration cost = 5,500
so merge joint does not pay off!

Example 1(d) (Cont.)

But say R1 = 10,000 blocks contiguous
R2 = 5,000 blocks not ordered

<u>Iterate:</u> $\frac{5000}{100} \times (100+10{,}000) = 50 \times 10{,}100$
$= 505{,}000$ IOs

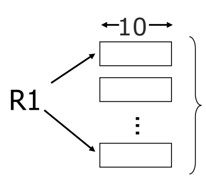
<u>Merge join:</u> 5(10,000+5,000) = 75,000 IOs

Merge Join (with sort) WINS!

Merge Sort

How much memory do we need for merge sort?

E.g: Say I have 10 memory blocks



100 chunks ⇒ to merge,
need 100 blocks!

In General

Say k blocks in memory
x blocks for relation sort

chunks = (x/k)
size of chunk = k

chunks $\leq$ buffers available for merge

so... $(x/k) \leq k$
or $k^2 \geq x$ or $k \geq \sqrt{x}$

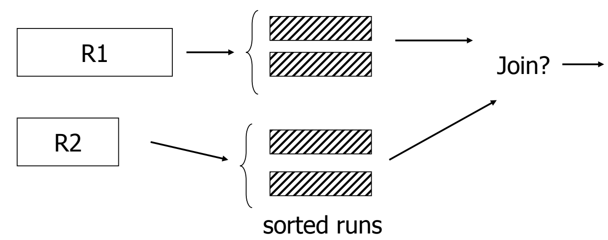
In Our Example

R1 is 1000 blocks, k ≥ 31.62
R2 is 500 blocks, k ≥ 22.36

Need at least 32 buffers

Can we improve on merge join?

Hint: do we really need the fully sorted files?



Cost of Improved Merge Join

C = Read R1 + write R1 into runs
+ read R2 + write R2 into runs
+ join
= 2000 + 1000 + 1500 = 4500

→ Memory requirement?

Example 1(e)

Index Join

- Assume R1.C index exists; 2 levels
- Assume R2 contiguous, unordered

- Assume R1.C index fits in memory

Example 1(e) (Cont.)

Cost: Reads: 500 IOs
for each R2 tuple:
- probe index - free
- if match, read R1 tuple: 1 IO

Example 1(e) (Cont.)

What is expected # of matching tuples?

(a) say R1.C is key, R2.C is foreign key
then expect = 1

(b) say V(R1,C) = 5000, T(R1) = 10,000
with uniform assumption
expect = 10,000/5,000 = 2

Total Cost with Index Join

(a) Total cost = 500+5000(1)1 = 5,500

(b) Total cost = 500+5000(2)1 = 10,500

What if index does not fit in memory?

Example: say R1.C index is 201 blocks

- Keep root + 99 leaf nodes in memory
- Expected cost of each probe is

$$E = (0)\frac{99}{200} + (1)\frac{101}{200} \approx 0.5$$

Total Cost (including probes)

= 500 + 5000 [Probe + get records]
= 500 + 5000 [0.5 + 2]   uniform assumption
= 500 + 12,500 = 13,000   (case b)

So Far

not contiguous	Iterate R2 $\bowtie$ R1	55,000 (best)
	Merge Join	________
	Sort+ Merge Join	________
	R1.C Index	________
	R2.C Index	________
contiguous	Iterate R2 $\bowtie$ R1	5500
	Merge join	1500
	Sort+Merge Join	7500 → 4500
	R1.C Index	5500
	R2.C Index	________

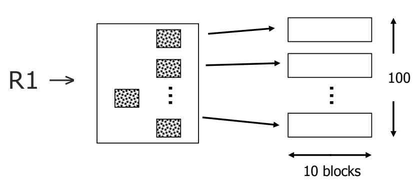
Example 1(f)

Hash Join

- R1, R2 contiguous (un-ordered)

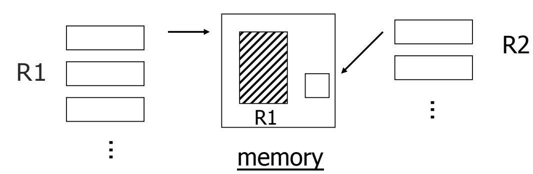
→ Use 100 buckets

→ Read R1, hash, + write buckets



Example 1(f) (Cond.)

→ Same for R2
→ Read one R1 bucket; build memory hash table
→ Read corresponding R2 bucket + hash probe



→ Then repeat for all buckets

Cost

- "Bucketize:" Read R1 + write
  Read R2 + write
- Join: Read R1, R2

- Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Minimum Memory Requirements

- Size of R1 bucket = (x/k)
  k = number of memory buffers
  x = number of R1 blocks

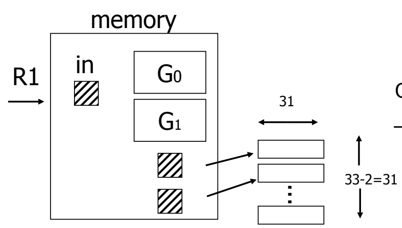
- So... (x/k) < k

- k > $\sqrt{x}$ need: k+1 total memory buffers

Trick

Keep Some Buckets in Memory

E.g., k'=33 R1 buckets = 31 blocks
keep 2 in memory



Memory use:

G0	31 buffers
G1	31 buffers
Output	33-2 buffers
R1 input	1 buffer
Total	94 buffers

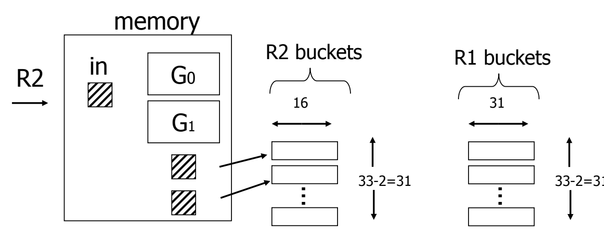
6 buffers to spare!!

called hybrid hash-join

Next

Bucketize R2

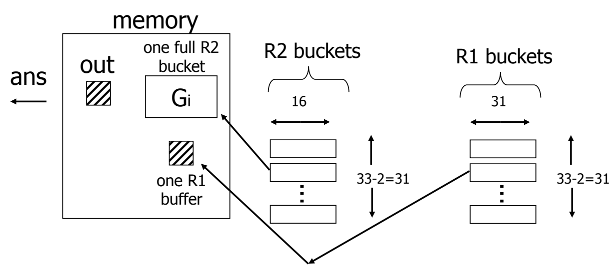
- R2 buckets =500/33= 16 blocks
- Two of the R2 buckets joined immediately with G0,G1



Finally

Join remaining buckets

- for each bucket pair:
  - read one of the buckets into memory
  - join with second bucket

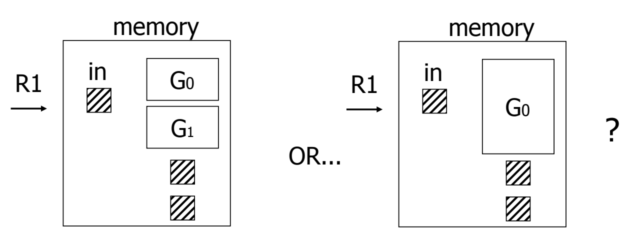


Cost

- Bucketize R1 = 1000 + 31 × 31 = 1961
- To bucketize R2, only write 31 buckets:
  so, cost = 500 + 31 × 16 = 996
- To compare join (2 buckets already done)
  read 31 × 31 + 31 × 16 = 1457

Total cost = 1961 + 996 + 1457 = 4414

How Many Buckets in Memory?



→ See textbook for answer...

Another Hash Join Trick

- Only write into buckets
  <val,ptr> pairs
- When we get a match in join phase,
  must fetch tuples

Another Hash Join Trick (Cont.)

- To illustrate cost computation, assume:
  - 100 <val,ptr> pairs/block
  - expected number of result tuples is 100
- Build hash table for R2 in memory
  5000 tuples → 5000/100 = 50 blocks
- Read R1 and match
- Read ~ 100 R2 tuples
- <u>Total cost</u> =

Read R2:	500	
Read R1:	1000	
Get tuples:	100	
	1600	

So Far

contiguous

Iterate	5500
Merge join	1500
Sort+merge joint	7500
R1.C index	5500 → 550
R2.C index	_____
Build R.C index	_____
Build S.C index	_____
Hash join	4500+
with trick,R1 first	4414
with trick,R2 first	_____
Hash join, pointers	1600

Summary

- Iteration ok for "small" relations
  (relative to memory size)
- For equi-join, where relations not sorted and no indexes exist, <u>hash join</u> usually best
- Sort + merge join good for non-equi-join (e.g., R1.C > R2.C)
- If relations already sorted, use merge join
- If index exists, it <u>could</u> be useful
  (depends on expected result size)