



## Lecture 32 of 42

### MySQL Primer 1 of 2, Fast Joins Concluded Discussion: DB Norm., Practical DBs

Monday, 06 November 2006

William H. Hsu  
Department of Computing and Information Sciences, KSU

KSOL course page: <http://snipurl.com/va60>

Course web site: <http://www.kddresearch.org/Courses/Fall-2006/CIS560>

Instructor home page: <http://www.cis.ksu.edu/~bhsu>

Reading for Next Class:  
MySQL 5.1 documentation



## Chapter 13: Query Processing

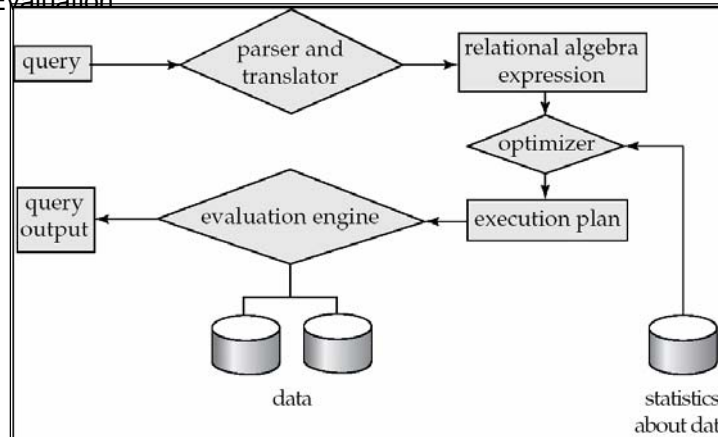
- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions





## Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



## Measures of Query Cost: Review

- Cost is generally measured as total elapsed time for answering query
  - \* Many factors contribute to time cost
    - ⇒ *disk accesses, CPU, or even network communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - \* Number of **seeks** \* average-seek-cost
  - \* Number of **blocks read** \* average-block-read-cost
  - \* Number of **blocks written** \* average-block-write-cost
    - ⇒ Cost to write a block is greater than cost to read a block
      - ◆ data is read back after being written to ensure that the write was successful



## Measures of Query Cost (Cont.)

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - \*  $t_T$  – time to transfer one block
  - \*  $t_S$  – time for one seek
  - \* Cost for  $b$  block transfers plus  $S$  seeks  
 $b * t_T + S * t_S$
- We ignore CPU costs for simplicity
  - \* Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae
- Several algorithms can reduce disk IO by using extra buffer space
  - \* Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - ⇒ We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
  - \* But hard to take into account for cost estimation



## External Sort-Merge

Let  $M$  denote memory size (in pages).

### 1. Create sorted runs. Let $i$ be 0 initially.

Repeatedly do the following till the end of the relation:

- Read  $M$  blocks of relation into memory
- Sort the in-memory blocks
- Write sorted data to run  $R_i$ ; increment  $i$ .

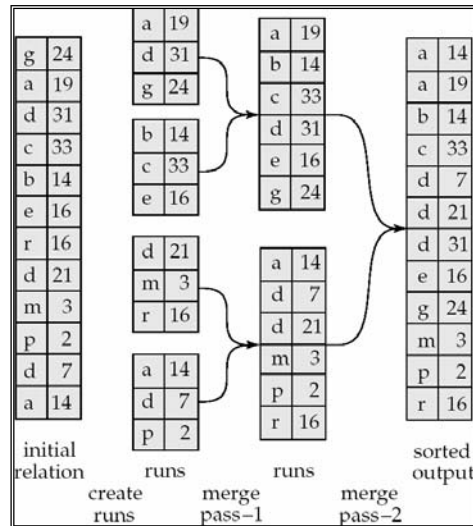
Let the final value of  $i$  be  $N$

### 2. Merge the runs (next slide).....





## Example: External Sorting Using Sort-Merge



## External Merge Sort (Cont.)

- Cost analysis:
  - \* Total number of merge passes required:  $\lceil \log_{M-1}(b_r/M) \rceil$ .
  - \* Block transfers for initial run creation as well as in each pass is  $2b_r$ 
    - ⇒ for final pass, we don't count write cost
      - ◆ we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
    - ⇒ Thus total number of block transfers for external sorting:
 
$$b_r (2 \lceil \log_{M-1}(b_r/M) \rceil + 1)$$
  - \* Seeks: next slide



## External Merge Sort (Cont.)

- Cost of seeks
  - \* During run generation: one seek to read each run and one seek to write each run
    - ⇒  $2 \lceil b_r / M \rceil$
  - \* During the merge phase
    - ⇒ Buffer size:  $b_b$  (read/write  $b_b$  blocks at a time)
    - ⇒ Need  $2 \lceil b_r / b_b \rceil$  seeks for each merge pass
      - ◆ except the final one which does not require a write
    - ⇒ Total number of seeks:  
 $2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{M-1}(b_r / M) \rceil - 1)$



## Join Operation

- Several different algorithms to implement joins
  - \* Nested-loop join
  - \* Block nested-loop join
  - \* Indexed nested-loop join
  - \* Merge-join
  - \* Hash-join
- Choice based on cost estimate
- Examples use the following information
  - \* Number of records of *customer*: 10,000    *depositor*: 5000
  - \* Number of blocks of *customer*: 400    *depositor*: 100



## Nested-Loop Join

- To compute the theta join  $\bowtie_{r \theta s}$ 
    - for each tuple  $t_r$  in  $r$  do begin**
      - for each tuple  $t_s$  in  $s$  do begin**
        - test pair  $(t_r, t_s)$  to see if they satisfy the join condition  $\theta$
        - if they do, add  $t_r \bullet t_s$  to the result.
    - end**
  - end**
- $r$  is called the **outer relation** and  $s$  the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.



## Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  - $n_r * b_s + b_r$
  - block transfers, plus
  - $n_r + b_r$
  - seeks
- If the smaller relation fits entirely in memory, use that as the inner relation.
  - \* Reduces cost to  $b_r + b_s$  block transfers and 2 seeks
- Assuming worst case memory availability cost estimate is
  - \* with *depositor* as outer relation:
    - $\Rightarrow 5000 * 400 + 100 = 2,000,100$  block transfers,
    - $\Rightarrow 5000 + 100 = 5100$  seeks
  - \* with *customer* as the outer relation
    - $\Rightarrow 10000 * 100 + 400 = 1,000,400$  block transfers and 10,400 seeks
- If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.





## Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```

for each block  $B_r$  of  $r$  do begin
  for each block  $B_s$  of  $s$  do begin
    for each tuple  $t_r$  in  $B_r$  do begin
      for each tuple  $t_s$  in  $B_s$  do begin
        Check if  $(t_r, t_s)$  satisfy the join condition
        if they do, add  $t_r \bullet t_s$  to the result.
      end
    end
  end
end

```



## Block Nested-Loop Join (Cont.)

- Worst case estimate:  $b_r * b_s + b_r$  block transfers +  $2 * b_r$  seeks
  - \* Each block in the inner relation  $s$  is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation)
- Best case:  $b_r + b_s$  block transfers + 2 seeks.
- Improvements to nested loop and block nested loop algorithms:
  - \* In block nested-loop, use  $M - 2$  disk blocks as blocking unit for outer relations, where  $M$  = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - ⇒ Cost =  $\lceil b_r / (M-2) \rceil * b_s + b_r$  block transfers +  $2 \lceil b_r / (M-2) \rceil$  seeks
  - \* If equi-join attribute forms a key or inner relation, stop inner loop on first match
  - \* Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - \* Use index on inner relation if available (next slide)





## Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - \* join is an equi-join or natural join and
  - \* an index is available on the inner relation's join attribute
    - ⇒ Can construct an index just to compute a join.
- For each tuple  $t_r$  in the outer relation  $r$ , use the index to look up tuples in  $s$  that satisfy the join condition with tuple  $t_r$ .
- Worst case: buffer has space for only one page of  $r$ , and, for each tuple in  $r$ , we perform an index lookup on  $s$ .
- Cost of the join:  $b_r(t_r + t_s) + n_r * c$ 
  - \* Where  $c$  is the cost of traversing index and fetching all matching  $s$  tuples for one tuple of  $r$
  - \*  $c$  can be estimated as cost of a single selection on  $s$  using the join condition.
- If indices are available on join attributes of both  $r$  and  $s$ , use the relation with fewer tuples as the outer relation.



## Example of Nested-Loop Join Costs

- Compute  $\bowtie$   $depositor \bowtie customer$ , with  $depositor$  as the outer relation.
- Let  $customer$  have a primary B+-tree index on the join attribute  $customer-name$ , which contains 20 entries in each index node.
- Since  $customer$  has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- $depositor$  has 5000 tuples
- Cost of block nested loops join
  - \*  $400 * 100 + 100 = 40,100$  block transfers +  $2 * 100 = 200$  seeks
    - ⇒ assuming worst case memory
    - ⇒ may be significantly less with more memory
- Cost of indexed nested loops join
  - \*  $100 + 5000 * 5 = 25,100$  block transfers and seeks.
  - \* CPU cost likely to be less than that for block nested loops join





## Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
  1. Join step is similar to the merge stage of the sort-merge algorithm.
  2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
  3. Detailed algorithm in book

	a1	a2
$pr$	a	3
	b	1
	d	8
	d	13
	f	7
	m	5
	q	6

	a1	a3
$ps$	a	A
	b	G
	c	L
	d	N
	m	B

$r$                        $s$



## Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is:
 
$$b_r + b_s \text{ block transfers} + \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil \text{ seeks}$$
  - \* + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B<sup>+</sup>-tree index on the join attribute
  - \* Merge the sorted relation with the leaf entries of the B<sup>+</sup>-tree .
  - \* Sort the result on the addresses of the unsorted relation's tuples
  - \* Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - ⇒ Sequential scan more efficient than random lookup

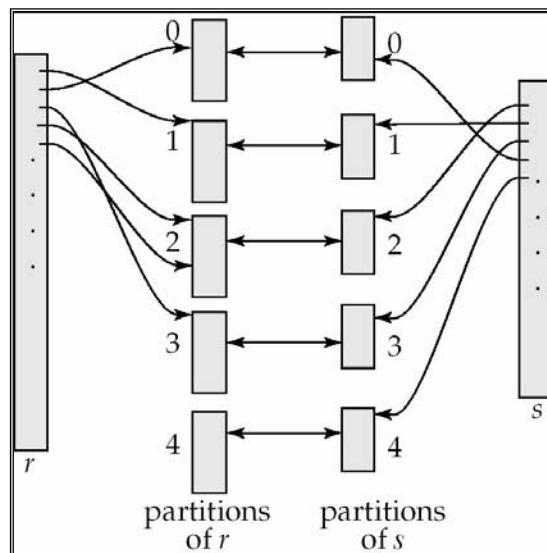


## Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function  $h$  is used to partition tuples of both relations
- $h$  maps *JoinAttrs* values to  $\{0, 1, \dots, n\}$ , where *JoinAttrs* denotes the common attributes of  $r$  and  $s$  used in the natural join.
  - \*  $r_0, r_1, \dots, r_n$  denote partitions of  $r$  tuples
    - ⇒ Each tuple  $t_r \in r$  is put in partition  $r_i$  where  $i = h(t_r[\text{JoinAttrs}])$ .
  - \*  $s_0, s_1, \dots, s_n$  denotes partitions of  $s$  tuples
    - ⇒ Each tuple  $t_s \in s$  is put in partition  $s_i$  where  $i = h(t_s[\text{JoinAttrs}])$ .
- Note: In book,  $r_i$  is denoted as  $H_{ri}$ ,  $s_j$  is denoted as  $H_{sj}$  and  $n$  is denoted as  $n_h$ .



## Hash-Join (Cont.)





## Hash-Join (Cont.)

- $r$  tuples in  $r_i$  need only to be compared with  $s$  tuples in  $s_j$   
Need not be compared with  $s$  tuples in any other partition, since:
  - \* an  $r$  tuple and an  $s$  tuple that satisfy the join condition will have the same value for the join attributes.
  - \* If that value is hashed to some value  $i$ , the  $r$  tuple has to be in  $r_i$  and the  $s$  tuple in  $s_i$ .



## Hash-Join Algorithm

The hash-join of  $r$  and  $s$  is computed as follows.

1. Partition the relation  $s$  using hashing function  $h$ . When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition  $r$  similarly.
3. For each  $i$ :
  - (a) Load  $s_i$  into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one  $h$ .
  - (b) Read the tuples in  $r_i$  from the disk one by one. For each tuple  $t_r$ , locate each matching tuple  $t_s$  in  $s_i$  using the in-memory hash index. Output the concatenation of their attributes.

Relation  $s$  is called the **build input** and  $r$  is called the **probe input**.





## Hash-Join algorithm (Cont.)

- The value  $n$  and the hash function  $h$  is chosen such that each  $s_i$  should fit in memory.
  - \* Typically  $n$  is chosen as  $\lceil b_s/M \rceil * f$  where  $f$  is a “fudge factor”, typically around 1.2
  - \* The probe relation partitions  $s_j$  need not fit in memory
- **Recursive partitioning** required if number of partitions  $n$  is greater than number of pages  $M$  of memory.
  - \* instead of partitioning  $n$  ways, use  $M - 1$  partitions for  $s$
  - \* Further partition the  $M - 1$  partitions using a different hash function
  - \* Use same partitioning method on  $r$
  - \* Rarely required: e.g., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with block size of 4KB.



## Handling of Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition  $s_i$  if  $s_i$  does not fit in memory. Reasons could be
  - \* Many tuples in  $s$  with same value for join attributes
  - \* Bad hash function
- **Overflow resolution** can be done in build phase
  - \* Partition  $s_i$  is further partitioned using different hash function.
  - \* Partition  $r_j$  must be similarly partitioned.
- **Overflow avoidance** performs partitioning carefully to avoid overflows during build phase
  - \* E.g. partition build relation into many partitions, then combine them
- Both approaches fail with large numbers of duplicates
  - \* Fallback option: use block nested loops join on overflowed partitions





## Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
 
$$3(b_r + b_s) + 4 * n_h \text{ block transfers} +$$

$$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) \text{ seeks}$$
- If recursive partitioning required:
  - \* number of passes required for partitioning build relations is  $\lceil \log_{M-1}(b_s) - 1 \rceil$
  - \* best to choose the smaller relation as the build relation.
  - \* Total cost estimate is:
 
$$2(b_r + b_s \lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s) \text{ block transfers} +$$

$$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) \lceil \log_{M-1}(b_s) - 1 \rceil \text{ seeks}$$
- If the entire build input can be kept in main memory no partitioning is required
  - \* Cost estimate goes down to  $b_r + b_s$ .

