



## Lecture 31 of 42

### Fast Joins (Continued) Discussion: DB Impl., Normalization Review

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:

Second half of Chapter 13, Silberschatz *et al.*, 5<sup>th</sup> edition



## Chapter 13: Query Processing

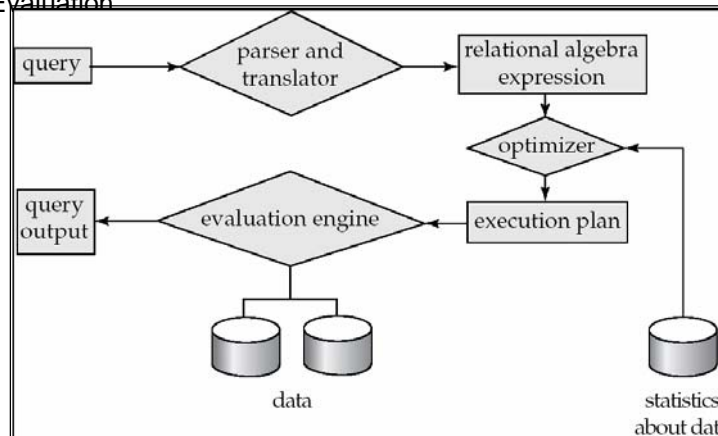
- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions





## 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



## Select [1]: Linear Search Review

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- **Algorithm A1 (linear search)**. Scan each file block and test all records to see whether they satisfy the selection condition.
  - \* **Cost estimate =  $b_r$  block transfers + 1 seek**
    - ⇒  $b_r$  denotes number of blocks containing records from relation  $r$
  - \* **If selection is on a key attribute, can stop on finding record**
    - ⇒ **cost =  $(b_r/2)$  block transfers + 1 seek**
  - \* **Linear search can be applied regardless of**
    - ⇒ selection condition or
    - ⇒ ordering of records in the file, or
    - ⇒ availability of indices





## Select [2]: Binary Search Review

- **A2 (binary search).** Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - \* Assume that the blocks of a relation are stored contiguously
  - \* Cost estimate (number of disk blocks to be scanned):
    - ⇒ cost of locating the first tuple by a binary search on the blocks
      - ◆  $\lceil \log_2(b_p) \rceil * (t_T + t_S)$
    - ⇒ If there are multiple records satisfying selection
      - ◆ Add transfer cost of the number of blocks containing records that satisfy selection condition
      - ◆ Will see how to estimate this cost in Chapter 14



## Selections Using Indices

- Index scan – search algorithms that use an index
  - \* selection condition must be on search-key of index.
- **A3 (primary index on candidate key, equality).** Retrieve a single record that satisfies the corresponding equality condition
  - \* Cost =  $(h_i + 1) * (t_T + t_S)$
- **A4 (primary index on nonkey, equality)** Retrieve multiple records.
  - \* Records will be on consecutive blocks
    - ⇒ Let  $b$  = number of blocks containing matching records
  - \* Cost =  $h_i * (t_T + t_S) + t_S + t_T * b$
- **A5 (equality on search-key of secondary index).**
  - \* Retrieve a single record if the search-key is a candidate key
    - ⇒ Cost =  $(h_i + 1) * (t_T + t_S)$
  - \* Retrieve multiple records if search-key is not a candidate key
    - ⇒ each of  $n$  matching records may be on a different block
    - ⇒ Cost =  $(h_i + n) * (t_T + t_S)$ 
      - ◆ Can be very expensive!





## Selections Involving Comparisons

- Can implement selections of the form  $\sigma_{A \leq v}(r)$  or  $\sigma_{A \geq v}(r)$  by using
  - \* a linear file scan or binary search,
  - \* or by using indices in the following ways:
- **A6 (primary index, comparison).** (Relation is sorted on A)
  - ⇒ For  $\sigma_{A \geq v}(r)$  use index to find first tuple  $\geq v$  and scan relation sequentially from there
  - ⇒ For  $\sigma_{A \leq v}(r)$  just scan relation sequentially till first tuple  $> v$ ; do not use index
- **A7 (secondary index, comparison).**
  - ⇒ For  $\sigma_{A \geq v}(r)$  use index to find first index entry  $\geq v$  and scan index sequentially from there, to find pointers to records.
  - ⇒ For  $\sigma_{A \leq v}(r)$  just scan leaf pages of index finding pointers to records, till first entry  $> v$
  - ⇒ In either case, retrieve records that are pointed to
    - ◆ requires an I/O for each record
    - ◆ Linear file scan may be cheaper



## Implementation of Complex Selections

- **Conjunction:**  $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$
- **A8 (conjunctive selection using one index).**
  - \* Select a combination of  $\theta_i$  and algorithms A1 through A7 that results in the least cost for  $\sigma_{\theta_i}(r)$ .
  - \* Test other conditions on tuple after fetching it into memory buffer.
- **A9 (conjunctive selection using multiple-key index).**
  - \* Use appropriate composite (multiple-key) index if available.
- **A10 (conjunctive selection by intersection of identifiers).**
  - \* Requires indices with record pointers.
  - \* Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - \* Then fetch records from file
  - \* If some conditions do not have appropriate indices, apply test in memory.





## Algorithms for Complex Selections

- **Disjunction:**  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ .
- **A11 (disjunctive selection by union of identifiers).**
  - \* Applicable if *all* conditions have available indices.
    - ⇒ Otherwise use linear scan.
  - \* Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - \* Then fetch records from file
- **Negation:**  $\sigma_{\neg\theta}(r)$ 
  - \* Use linear scan on file
  - \* If very few records satisfy  $\neg\theta$ , and an index is applicable to  $\theta$ 
    - ⇒ Find satisfying records using index and fetch from file



## Sorting

- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, **external sort-merge** is a good choice.





## 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:

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

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

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



## External Sort-Merge (Cont.)

**2. Merge the runs (N-way merge).** We assume (for now) that  $N < M$ .

1. Use  $N$  blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page

**2. repeat**

1. Select the first record (in sort order) among all buffer pages
2. Write the record to the output buffer. If the output buffer is full write it to disk.
3. Delete the record from its input buffer page.  
**If** the buffer page becomes empty **then** read the next block (if any) of the run into the buffer.

**3. until** all input buffer pages are empty:

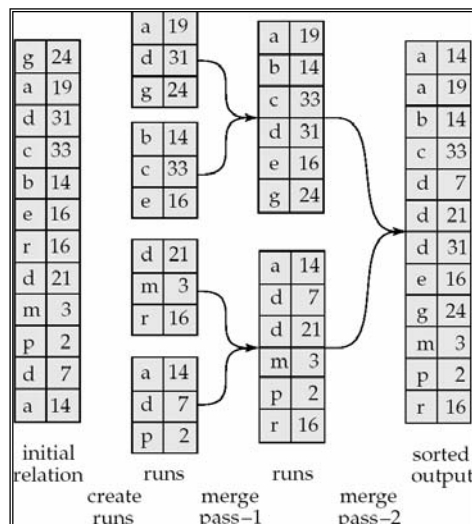


## External Sort-Merge (Cont.)

- If  $N \geq M$ , several merge passes are required.
  - \* In each pass, contiguous groups of  $M - 1$  runs are merged.
  - \* A pass reduces the number of runs by a factor of  $M - 1$ , and creates runs longer by the same factor.
    - ⇒ E.g. If  $M=11$ , and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - \* Repeated passes are performed till all runs have been merged into one.



## 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  $\bowtie_{\theta} r \text{ } s$   
**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:
    - ⇒  $5000 * 400 + 100 = 2,000,100$  block transfers,
    - ⇒  $5000 + 100 = 5100$  seeks
  - \* with *customer* as the outer relation
    - ⇒  $10000 * 100 + 400 = 1,000,400$  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  $depositor \bowtie customer$ , with *depositor* as the outer relation.
- Let *customer* have a primary B<sup>+</sup>-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
  - \*  $400 \cdot 100 + 100 = 40,100$  block transfers +  $2 \cdot 100 = 200$  seeks
    - ⇒ assuming worst case memory
    - ⇒ may be significantly less with more memory
- Cost of indexed nested loops join
  - \*  $100 + 5000 \cdot 5 = 25,100$  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

$pr$	a1	a2	$ps$	a1	a3
→	a	3	→	a	A
	b	1		b	G
	d	8		c	L
	d	13		d	N
	f	7		m	B
	m	5			
	q	6			s
		$r$			





## 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:  
 $b_r + b_s$  block transfers +  $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$  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



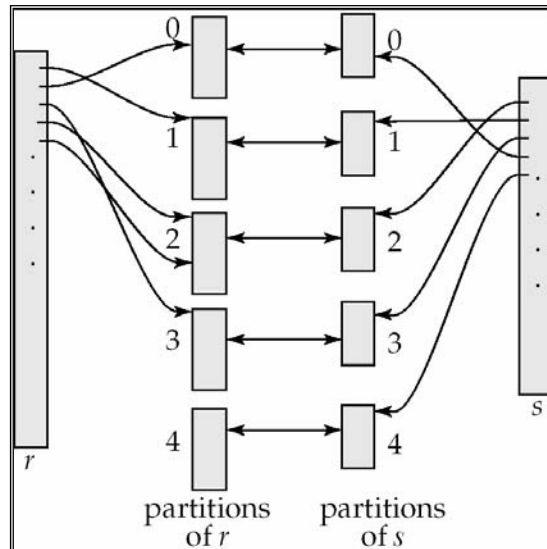
## Hash-Join

- 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  $i = h(t_r[\text{JoinAttrs}])$ .
  - \*  $r_0, r_1, \dots, r_n$  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,  $r_j$  is denoted as  $H_{rj}$ ,  $s_j$  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  $s_i$   
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  $s_i$  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  $r_i$  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 relation  $s$  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  $b_r + b_s$ .

