

# Databases Illuminated

## Chapter 11

### Query Optimization

# Query processing overview

- Steps in executing SQL query-DBMS:
  - Checks query syntax
  - Validates query-checks data dictionary; verifies objects referred to are database objects and requested operations are valid
  - Translates query into relational algebra (or relational calculus)
  - Rearranges relational algebra operations into most efficient form
  - Uses its knowledge of table size, indexes, order of tuples, distribution of values, to determine how the query will be processed-estimates the "cost" of alternatives and chooses the plan with the least estimated cost-considers the number of disk accesses, amount of memory, processing time, and communication costs, if any
  - Execution plan is then coded and executed
  - **Figure 11.1** summarizes this process

# Relational Algebra Translation

- SQL Select..From..Where usually translates into combination of RA SELECT, PROJECT, JOIN
- RA SELECT-unary operator:  $\sigma_p(\textit{table-name})$ 
  - p is a predicate, called the  $\sigma$  (theta) condition
  - Returns entire rows that satisfy  $\sigma$
- RA PROJECT-unary operator:  $\pi_{\textit{proj-list}}(\textit{table-name})$ 
  - Proj-list is a list of columns
  - Returns unique combinations of values for those columns
- RA JOIN-binary operator:  $\textit{table1} \bowtie \textit{table2}$ 
  - Compares table1 and table2, which have a common column (or columns with same domain)
  - Chooses rows from each that match on common column
  - Combines those rows, but shows common column only once

# Query Tree

- Graphical representation of the operations and operands in relational algebra expression
- Leaf nodes are relations
- Unary or binary operations are internal nodes
- An internal node can be executed when its operands are available
- Node is replaced by the result of the operation it represents
- Root node is executed last, and is replaced by the result of the entire tree
- See **Figure 11.2**

# Doing SELECT early

- Same SQL statement can be translated to different relational algebra statements
- Performing SELECT early reduces size of intermediate nodes-See **Figure 11.2(b)**
- Push SELECT as far down the tree as possible
- For conjunctive SELECT, do each part on its own tree instead of waiting for join-See **Figure 11.3(b)**

# Some Properties of Natural JOIN

- Associative
  - (Student |x| Enroll) |x| Class same as
  - Student |x| (Enroll |x| Class)
- Commutative, ignoring column order
  - Enroll |x| Class ? Class |x| Enroll

Many similar rules exist

# RA Equivalences-1

1. *All joins and products are commutative.*

$R \bowtie S \bowtie S \bowtie R$  and

$R \Join S \Join S \Join R$  and

$R \Join S \bowtie S \bowtie R$

2. *Joins and products are associative*

$(R \bowtie S) \bowtie T \bowtie R \bowtie (S \bowtie T)$

$(R \Join S) \Join T \bowtie R \Join (S \Join T)$

$(R \Join S) \bowtie T \bowtie R \Join (S \bowtie T)$

# RA Equivalences-2

3. *Select is commutative*

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

4. *Conjunctive selects can cascade into individual selects*

$$\sigma_{p\&q\&\dots\&z}(R) = (\sigma_p(\sigma_q(\dots(\sigma_z(R))\dots)))$$

5. *Successive projects can be reduced to the final project.*

If list1, list2, ..., listn are lists of attribute names and each of the list<sub>i</sub> contains list<sub>i-1</sub>, then

$$\sigma_{list1}(\sigma_{list2}(\dots\sigma_{listn}(R)\dots)) = \sigma_{list1}(R)$$

- So only the last project has to be executed

# RA Equivalences-3

## 6. *Select and project sometimes commute*

- If p involves only the attributes in projlist, then select and project commute

$$\pi_{\text{projlist}} (\sigma_p (R)) = \sigma_p (\pi_{\text{projlist}} (R))$$

## 7. *Select and join (or product) sometimes commute*

- If p involves only attributes of one of the tables being joined, then select and join commute

$$\sigma_p (R \bowtie S) = (\sigma_p (R)) \bowtie S$$

- Only if p refers just to R

# RA Equivalences-4

## 8. *Select sometimes distributes over join (or product)*

- For  $p$  AND  $q$ , where  $p$  involves only the attributes of  $R$  and  $q$  only the attributes of  $S$  the select distributes over the join

$$\sigma_{p \text{ AND } q} (R \bowtie S) = (\sigma_p (R)) \bowtie (\sigma_q (S))$$

## 9. *Project sometimes distributes over join (or product)*

- If projlist can be split into separate lists, list1 and list2, so that list1 contains only attributes of  $R$  and list2 contains only attributes of  $S$ , then

$$\sigma_{\text{projlist}} (R \bowtie S) = (\sigma_{\text{list1}} (R)) \bowtie (\sigma_{\text{list2}} (S))$$

# RA Equivalences-5

10. *Union and intersection are commutative*

$$R \cup S = S \cup R$$

$$R \cap S = S \cap R$$

- Set difference is not commutative.

11. *Union and intersection are individually associative.*

$$(R \cup S) \cup T = R \cup (S \cup T)$$

$$(R \cap S) \cap T = R \cap (S \cap T)$$

- Set difference is not associative

# RA Equivalences-6

**12. Select distributes over union, intersection, and difference**

$$\begin{aligned} \sigma_p(R \cup S) &= \sigma_p(R) \cup \sigma_p(S) \\ \sigma_p(R \cap S) &= \sigma_p(R) \cap \sigma_p(S) \\ \sigma_p(R - S) &= \sigma_p(R) - \sigma_p(S) \end{aligned}$$

**13. Project distributes over union, intersection, and difference**

$$\begin{aligned} \pi_{projlist}(R \cup S) &= (\pi_{projlist}(R)) \cup (\pi_{projlist}(S)) \\ \pi_{projlist}(R \cap S) &= (\pi_{projlist}(R)) \cap (\pi_{projlist}(S)) \\ \pi_{projlist}(R - S) &= (\pi_{projlist}(R)) - (\pi_{projlist}(S)) \end{aligned}$$

**14. Project is idempotent-repeating it produces the same result**

$$\pi_{projlist}(\pi_{projlist}(R)) = \pi_{projlist}(R)$$

**15. Select is idempotent**

$$\sigma_p(\sigma_p(R)) = \sigma_p(R)$$

# Heuristics for Optimization

- **Do selection as early as possible.** Use cascading, commutativity, and distributivity to move selection as far down the query tree as possible
- Use associativity to rearrange relations so the selection operation that will produce the smallest table will be executed first
- If a product appears as an argument for a selection, where the selection involves attributes of the tables in the product, change the product to a join
  - If the selection involves attributes of only one of the tables in the product, apply the selection to that table first
- **Do projection early.** Use cascading, distributivity and commutativity to move the projection as far down the query tree as possible.
- Examine all projections to see if some are unnecessary
- If a sequence of selections and/or projections have the same argument, use commutativity or cascading to combine them into one selection, one projection, or a selection followed by a projection
- If a sub-expression appears more than once in the query tree, and the result it produces is not too large, compute it once and save it

# Cost Factors

- Cost factors of executing a query
  - Cost of reading files
  - Processing costs once data is in main memory
  - Cost of writing and storing intermediate results
  - Communication costs
  - Cost of writing final results to storage
- Most significant factor is the number of disk accesses, the read and write costs
- System uses statistics stored in the data dictionary and knowledge about the size, structure and access methods of each file

# Estimating Access Cost

- **Access cost**-number of blocks brought into main memory for reading or written to secondary storage as result
- Tables may be stored in
  - **packed form**-blocks contain only tuples from one table
  - **unpacked form**, tuples are interspersed with tuples from other tables
- If unpacked, have to assume every tuple of relation is in a different block
- If packed, estimate the number of blocks from tuple size, number of tuples, and capacity of the block
- Some useful symbols:
  - $t(R)$ , number of tuples in relation  $R$
  - $b(R)$ , number of blocks needed to store  $R$
  - $bf(R)$ , number of tuples of  $R$  per block, called **blocking factor** of  $R$
  - **If  $R$  is packed, then  $b(R) = t(R)/bf(R)$**
- Example: If the Student relation has blocks of 4K bytes, and there are 10,000 student records each 200 bytes long, then 20 records fit per block ( $4096/200$ ). We need  $10000/20$  or 500 blocks to hold this file in packed form

# Access Paths for a Table

- May be in order by key (primary or secondary)
- May be hashed on the value of a primary key
- May have an index on the primary key, and/or secondary indexes on non-primary key attributes

# Indexes

- May be **clustered index**, tuples with the same value of the index appear in the same block-one per table. Other indexes will then be **non-clustered**
- May be **dense**, having an entry for each tuple of the relation, or **non-dense**
- Normally **B+ tree** or a similar structure is used
- Must first access the index itself, so cost of accessing the index must be considered, in addition to data access
- Index access cost is usually small compared to the cost of accessing the data records

# Symbols for Cost of Using Indexes

- $l(\text{index-name})$ , the number of levels in a multi-level index, or the average number of index accesses needed to find an entry
- $n(A,R)$ , the number of distinct values of attribute  $A$  in relation  $R$
- If the values of  $A$  are **uniformly distributed** in  $R$ , then the number of tuples expected to have a particular value,  $c$ , for  $A$ , the **selection size** or  $s(A=c,R)$ , is
  - $s(A=c,R) = t(R)/n(A,R)$
  - if  $A$  is a candidate key, each tuple has a unique value for  $A$ , so  $n(A,R) = t(R)$  and the selection size is 1

# Example

- Estimate the number of students in the university with a major of Mathematics
  - If there are 10,000 students,  $t(\text{Student}) = 10000$
  - If there are 25 major subjects,  $n(\text{major}, \text{Student}) = 25$
  - Then number of Mathematics majors is

$$s(\text{major}='Math', \text{Student}) = t(\text{Student})/n(\text{major}, \text{Student}) = 10000/25 = 400$$

- We assume majors are uniformly distributed, that the number of students choosing each major is about equal
- Some systems use **histograms**, graphs that show the frequencies of different values of attributes. The histogram gives a more accurate estimate of the selection size for a particular value
- Some systems store the minimum and maximum values for each attribute

# Estimating Cost for SELECT, $?_{A=c}(R)$

- Depends on what access paths exist
  - If file hashed on the selection attribute(s)
  - If file has an index on the attribute(s) and whether the index is clustered
  - If file is in order by the selection attribute(s)
  - If none of the above applies

# Full Table Scan

- “Worst case” method-always compare other methods to this one
- Used when there is no access path for the attribute
- Cost is the number of blocks in the table-have to examine every tuple in the table to see if it qualifies
- Cost is  **$b(R)$**

Example, find all students who have first name of “Tom”.

We need to access each block of Student.

If the number of blocks of Student is  $10000/20$  or 500,

Reading Cost ( $?_{\text{firstName}='Tom'}(\text{Student})$ ) =  $b(\text{Student}) = 500$

# Using Hash Key

- Suppose A is a hash key having unique values
- Apply the hashing algorithm to calculate the target address for the record
- For no overflow, the expected number of accesses is 1
- If overflow, need an estimate of the average number of accesses required to reach a record, depends on the amount of overflow and the overflow handling method
- This statistic,  $h$ , may be available to the optimizer
- cost is  $h$

Example, suppose Faculty file is hashed on facId and  $h=2$

Reading Cost ( $?_{\text{facId}='F101'}(\text{Faculty})$ ) = 2

# Index on Unique Key

- For index on a unique key field, retrieve index blocks and then go directly to the record from the index
- System stores the number of levels in indexes
- Cost is  **$I(\text{index-name}) + 1$**

Example:  $?_{\text{stuld} = 'S1001'}$  (Student)

Since stuld is the primary key, suppose index on stuld is called Student\_stuld\_ndx, and has 3 levels

Reading Cost ( $?_{\text{stuld} = 'S1001'}$  (Student)) =  
 $I(\text{Student\_stuld\_ndx}) + 1 = 3 + 1 = 4$

# Non-clustered Index on a Secondary Key Attribute

- Suppose there is a non-clustering index on secondary key A
- Number of tuples that satisfy the condition is the selection size of the indexed attribute,  $s(A=c,R)$
- Must assume the tuples are on different blocks
- Assume all the tuples having value  $A=c$  are pointed to by the index, perhaps using a linked list or an array of pointers
- Cost is the number of index accesses plus the number of blocks for the tuples that satisfy the condition, or  
 **$I(\text{index-name}) + s(A=c,R)$**

Example, assume a non-clustering index on major in Student. For

?<sub>major='CSC'</sub> (Student), find records having a major value of 'CSC' by reading the index node for 'CSC' and going from there to each tuple it points to. If the index has 2 levels, cost is

Reading Cost(?<sub>major='CSC'</sub> (Student)) =  $I(\text{Student\_major\_ndx}) + s(\text{major='SCS', Student}) = 2 + (10000/25) = 2+400 = 402$

- Note that this is only slightly less than the worst case cost, which is 500.

# Selection Using a Clustered Index

- If we have a clustering index on A, use the selection size for A divided by the blocking factor to estimate the number of data blocks
- Assume the tuples of R having value A = c reside on contiguous blocks, so this calculation estimates the number of blocks needed to store these tuples
- We add that to the number of index blocks needed
- Cost is then  **$I(\text{index-name}) + (s(A=c,R))/bf(R)$**
- Example, if the index on major in the Student file were a clustering index, we would assume that the 400 records expected to have this value for major would be stored on contiguous blocks and the index would point to the first block. Then we could simply retrieve the following blocks to find all 400 records. The cost is

$$\text{Reading Cost}(\text{? major='CSC' (Student)}) = I(\text{Student\_major\_ndx}) + s(\text{major='SCS', Student})/bf(\text{Student}) = 2+400/20 = 22$$

# Selection on an Ordered File

- A is a key with unique values and records are in order by A
- Use binary search to access the record with A value of c
- Cost is approximately  $\log_2 b(R)$

Example, find a class record for a given classNumber, where Class file is in order by classNumber. Calculating the number of blocks in the table, if there are 2,500 Class records, each 100 bytes long, stored in blocks of size 4K, the blocking factor is  $4096/100$ , or 40, so the number of blocks is  $2500/40$  or 63

Reading Cost( $?_{\text{classNumber}='Eng201A'}$  (Class)) =  $\log_2(63) \sim 6$

- If A is not a key, may be several records with A value of c. Estimate must consider the selection size,  $s(A=c,R)$  divided by the number of records per block.
- Cost is  $\log_2 b(R) + s(A=c,R)/bf(R)$

# Conjunctive Selection with a Composite Index

- If predicate is a conjunction and a composite index exists for the attributes in the predicate, this case reduces to one of the previous cases
- Cost depends on whether the attributes are a composite key, and whether the index is clustered

# Conjunctive Selection without a Composite Index

- If one of the conditions involves an attribute which is used for ordering records in the file, or has an index or a hash key, then we use the appropriate method from those previously described to retrieve records that satisfy that part of the predicate, using the cost estimates given previously
- Once we retrieve the records we check to see if they satisfy the rest of the conditions
- If no attribute can be used for efficient retrieval, use the full table scan and check all the conditions simultaneously for each tuple

# Processing Joins

- The join is generally the most expensive operation to perform in a relational system
- Since it is often used in queries, it is important to be able to estimate its cost
- Cost depends on the method of processing as well as the size of the results

# Estimating Size of the Join Result-1

- Let R and S have size  $t(R)$  and  $t(S)$
- If the tables have no common attributes, can only do a Cartesian product, and the number of tuples in the result is  $t(R) * t(S)$
- If the set of common attributes is a key for one of the relations, the number of tuples in the join can be no larger than the number of tuples in the other relation, since each of these can match no more than one of the key values
- If the common attributes are a key for R, then the size of the join is **less than or equal to  $t(S)$**

Ex. For natural join of Student and Enroll, since stuld is the primary key of Student, the number of tuples in the result will be the same as the number of tuples in Enroll, or 50,000, since each Enroll tuple has exactly one matching Student tuple

# Estimating Size of the Join Result-2

- If common attributes are not a key of either relation
  - assume that there is one common attribute, A, whose values are uniformly distributed in both relations. For a particular value, c, of A in R, the number of tuples in S having a matching value of c for A is the selection size of A in S, or  $s(A=c, S)$ , which is  $t(S)/n(A, S)$ . This gives us the number of matches in S for a particular tuple in R. However, since there are  $t(R)$  tuples in R, each of which may have this number of matches, the total expected number of matches in the join is

$$t(R \bowtie S) = t(R) * t(S) / n(A, S)$$

- If we had started by considering tuples in S and looked for matches in R, we would have derived  $t(R \bowtie S) = t(S) * t(R) / n(A, R)$
- Use the formula that gives the smaller result

# Cost of Writing Join Result

- Estimate the number of bytes in the joined tuples to be roughly the sum of the bytes of R and S, and divide the block size by that number to get the blocking factor of the result
- The number of blocks in the result is the expected number of tuples divided by the blocking factor

# Methods of Performing Joins

- Nested loops-default method
- Sort-merge join
- Using index or hash key

# Cost of Performing Nested Loop Joins

- Default method, used when no special access paths exist
- If we have two buffers for reading, plus one for writing the result, bring the first block of R into the first buffer, and then bring each block of S, in turn, into the second buffer
- Compare each tuple of the current R block with each tuple of the current S block before switching in the next S block
- Once finished all the S blocks, bring in the next R block into the first buffer, and go through all the S blocks again
- Repeat this process until all of R has been compared with all of S
- See Figure 11.5
- **Read cost (R | S) =  $b(R) + (b(R)*b(S))$**

since each block of R has to be read, and each block of S has to be read once for each block of R.

# More on Nested Loop Joins

- Should pick the smaller file for the outside loop, since number of blocks in file in the outer loop file must be added to the product
- If buffer can hold more than three blocks, read as many blocks as possible from the outer loop file, and only one block from the inner loop file, plus one for writing the result
- If  $b(B)$  is the number of buffer blocks, using  $R$  as the outer loop, read  $b(B)-2$  blocks of  $R$  into the buffer at a time, and 1 block of  $S$
- Total number of  $R$  blocks still  $b(R)$ , but number of  $S$  blocks is approximately  $b(S) * (b(R) / (b(B) - 2))$
- The cost of accessing the files:  
 **$b(R) + ((b(S) * (b(R))) / (b(B) - 2))$**

# Nested Join with Small Files

- If all of  $R$  fits in main memory, with room for one block of  $S$  and one block for result, then read  $R$  only once, while switching in blocks of  $S$  one at a time
- The cost of reading the two packed files is then the most efficient possible cost  
 **$b(R) + b(S)$**

# Sort-Merge Join

- If both files are sorted on the attribute(s) to be joined, the join algorithm is like the algorithm for merging two sorted files
- **$b(R) + b(S)$**
- If unsorted, may be worthwhile to sort files before a join
- Add the cost of sorting, which depends on the sorting method used

# Join Using Index or Hash Key

- if A is a hash key for S, retrieve each tuple of R in the usual way, and use hashing algorithm to find all the matching records of S. Cost is

$$\mathbf{b(R) + t(R)*h}$$

- For index, cost depends on the type of index
- If A is the primary key of S, access cost is cost of accessing blocks of R plus cost of reading the index and accessing one record of S for each of the tuples in R

$$\mathbf{b(R) + (t(R) * (l(indexname) + 1))}$$

- If A is not a primary key, consider the number of matches in S per tuple of R,  $\mathbf{b(R) + (t(R) * (l(indexname) + s(A=c,S)))}$
- If the index is a clustering index, reduce the estimate by dividing by the blocking factor

$$\mathbf{b(R) + (t(R) * (l(indexname) + s(A=c,S)/bf(S))}$$

# Cost of Projection with Key

- Projection requires finding the values of the attributes in the projection list for each tuple, and eliminating duplicates, if any
- If the projection list contains a key of the relation, there are no duplicates to eliminate
  - The read cost is number of blocks in the relation is  **$b(R)$**
  - The number of tuples in the result is number of tuples in the relation,  $t(R)$
  - The resulting tuples may be much smaller than the tuples of  $R$ , so the number of blocks needed to write the result may be much smaller than  $b(R)$

# Cost of Projection-General Case

- If the projection list does not contain key, must eliminate duplicates
- Method Using Sorting
  - Sort the results so that duplicates appear together
  - eliminate any tuple that is a duplicate of the previous one
  - Cost is the sum of the costs of
    - Accessing all the blocks of the relation to create a temporary file with only attributes on the projection list
    - Writing the temporary file
    - Sorting the temporary file
    - Accessing the sorted temporary file to eliminate duplicates
    - Writing the final results file
  - Most expensive step is sorting temporary file
    - Use external sorting since DB files are large
    - Can use two-way merge sort can be used if there are 3 buffers available
    - if file has  $n$  pages, the number of passes needed will be  $(\log_2 n)+1$ , and the number of disk accesses required just for the sorting phase will be  **$2n((\log_2 n)+1)$**
- Can use hashing method if several buffers available

# Cost of Set Operations

- Sort both files on the same attributes
- Use basic sort-merge algorithm
  - For union, put in results file any tuple that appears in either of the original files, but drop duplicates
  - For intersection, place in the results file only the tuples that appear in both of the original files, but drop duplicates
  - For set difference,  $R - S$ , examine each tuple of  $R$  and place it in the results file if it has no match in  $S$
- Cost is the sum of the cost of
  - Accessing all the blocks of both files
  - Sorting both and writing the temporary sorted files
  - Accessing the temporary files to do the merge
  - Writing the results file

# Pipelining

- Materialization of intermediate results can be expensive
- **pipelining**
  - tuples “pass through” from one operation to the next in the pipeline, without creation of a temporary file
  - cannot be used in algorithms that require that the entire relation as input