CS352 Lecture - Query Processing / Optimization

Last revised 2/27/2017

Objectives:

1. To understand why the choice of strategy for performing a query can have a huge impact on processing time
2. To be familiar with various strategies for performing selection operations
3. To be familiar with various strategies for performing join operations
4. To be familiar with how statistical information can be used for evaluating strategies

Materials:

1. Projectables of pseudo-code for join strategies
2. Projectables of equivalence rules for queries (authors' powerpoints)

I. Introduction

A. Given a query, the DBMS must interpret it and "plan" a strategy for carrying it out. For all but the simplest queries, there will probably be several ways of doing so - with total processing effort possibly varying by several orders of magnitude.

B. Some of the issues have to do with the way that data is physically stored on disk.

Recall that, for most queries, the cost of accesses to data on disk far exceeds any other other cost, and becomes the most significant factor in determining the time needed to process most queries.

C. Some of the issues have to do with the fact that several different, but equivalent, formulations of a query may be possible - perhaps with vastly different execution costs.

Example: Given the schemes: BookAuthor(callNo, authorName)
BookTitle(callNo, title)

1. The question "Give me the titles of all books written by Korth" can be answered by either of the following two relational algebra queries:

   $\pi_{\text{title}} \quad \sigma_{\text{authorName = 'Korth'}} \quad \text{BookTitle |X| BookAuthor}$

   $\pi_{\text{title}} \quad \text{BookTitle |X|} \quad \sigma_{\text{authorName = 'Korth'}} \quad \text{BookAuthor}$

2. Because relational algebra is a procedural language, each of these queries implies a particular sequence of operations:

   a) The first query suggests that we should first do a natural join of BookAuthor and BookTitle, then select from it the tuples with authorName = 'Korth', then project out the title field.

   b) The second query suggests that we should first select only those tuples from Book Author where authorName = 'Korth', then join these with BookTitle, then project out the title field.

3. Suppose our database held 10,000 BookTitle tuples and 20,000 BookAuthor tuples (an average of two authors/book.) Suppose, further, that only 2 BookAuthor tuples contained 'Korth'. How would the cost of these two strategies compare?

   a) The first strategy would process all 10,000 BookTitle tuples at least once, and likewise all 20,000 BookAuthor tuples. Further, it would involve creating a temporary relation with 3 attributes (callNo, title, authorName) and (presumably) 20,000 tuples. Each of these would need to be examined to see if it pertains to 'Korth'. Thus, a total of 50,000 tuples would need to be processed - minimum. (And the minimum could only be achieved if the join is facilitated by an index on callNo for at least one of the two original schemes.)

b) On the other hand, the second strategy might involve processing only 2 BookAuthor tuples and the corresponding 2 BookTitle tuples (if appropriate indices exist) - for a total of 4 tuples. This is an effort ratio of 50,000: 4 = 12,500:1.

4. A low-performance DBMS might put the burden on the user to formulate his/her queries in such a way as to allow the most efficient processing of them. A good DBMS, however, will transform a given query into a more efficient equivalent one whenever possible.

   Example: If the first query above were given to a simple DBMS, it would perform very much less efficiently than if it were given the second. However, a sophisticated DBMS would transform the first query into something like the second before processing it if that were the form it was given.

D. In comparing ways to process a query, we focus on minimizing disk accesses, since disk access time is orders of magnitude greater than time for main memory accesses.

   Thus, it is often advantageous to do extra computation in memory to minimize disk accesses.

E. In the remainder of this series of lectures, we want to explore the following topics:

1. Strategies for performing selection

2. Strategies for performing joins

   Both of these, in turn, are influenced by issues such as the way the data is stored physically on the disk, and the availability of indices

3. Rules of equivalence for transforming queries

4. The use of statistical information to help evaluate query-processing strategies.

II. Selection Strategies

A. Consider a selection expression like

$$\sigma_{\text{SomeCondition}} \text{ SomeTable}$$

We consider several possibilities for the condition

1. It involves searching for an exact match for the value of some attribute (e.g. “borrowerID = '12345'”), and the attribute is a candidate key for the table, so we expect at most one match.

2. It involves searching for an exact match for the value of some attribute, but the attribute is <u>not</u> a candidate key for the table, so we can have any number of matches

3. It involves searching for a value of some attribute lying in some range (e.g.“`where age between 21 and 65`” or “`where age < 21`”),

4. It involves some more complex condition - perhaps involving a compound condition (“and” or “or”) or the values of two or more attributes.

B. One way to handle any selection condition is to use <u>linear search</u> - scan through all the tuples in the table, and find the ones that match.

If the tuples are blocked, then it suffices to perform one disk access for a block, and then scan the buffered copies of the records in turn.

Example: a table with 10,000 tuples - blocked 20 per block - would require 500 disk accesses - which would take on the order of 500*10ms = 5 seconds.

C. Linear search can often be avoided if the table has a relevant index.

1. Exact match queries can be facilitated if the table has an index on the attribute we are searching for. At this point, we need to consider a number of possibilities:

   a) The attribute on which the search is based is a superkey. In this case, searching the index will take us directly to the one and only tuple desired. This requires one block access plus whatever accesses are needed to use the index.

   b) The attribute on which the search is based is not a superkey - so there will likely be multiple matches.

      (1)If the index is clustering (the primary index for the table) , then the index will take us to the first matching tuple. Since the index is in key order, it is likely that other matches will be in the same block, or perhaps the next block - so 1 (or sometimes more) block accesses plus whatever accesses are needed to use the index.

      (2)If the index is non-clustering, then the index entry contains pointers to the relevant tuples, each of which is likely in a different block - so we need as many block accesses as there are matches plus whatever is needed to use the index.

2. If the query is a range query, an index will take us to the <u>first</u> tuple that satisfies the query. (We search for the starting value of the range, and the index takes us to the first tuple greater than or equal to this.)

   (1)If the index is clustering (the primary index for the table), then successive tuples will lie in the same or successive blocks. Each block containing a tuple that matches the query will be processed just once.

(2) If it is non-clustering but ordered (e.g. a B+ tree), then we can find successive matching tuples following pointers from successive index entries. The number of blocks read will be one per match found by the query.

(3)However, if it is not ordered (a hashtable) then the index doesn't really help in this case.

3. If the query has a more complicated structure (e.g. involves and, or, not), it may be possible to make use of indexes to create a list of pointers to tuples and then perform the computation on the lists before retrieving the actual tuples.

4. Of course, when estimating the cost of a selection using an index, we need to consider <u>both</u> the cost of accessing the relevant block(s) of the index and the cost of accessing the data.

   a) For example, if a table is stored as a B-Tree of height 3, then access to a piece of data using the index involves - in principle - three disk accesses.

   b) However, we will almost certainly keep a copy of the root of the tree in a buffer - reducing the number of accesses to two. Moreover, we may be able to buffer the second level blocks of the index as well - in which case an access using the index only involves accessing the data block.

   c) Even if we can't do this, though, use of an index will still beat linear search - 3 disk accesses is a lot less than retrieving every block!

III. Performing Joins Efficiently

A. Joins are the most expensive part of query processing, because the number of tuples examined to do a join can approach the product of the size of the two relations involved. Thus, the query optimizer must give considerable attention to choosing the best join strategy.

B. As a worst-case example, consider the following query:

$$\sigma_{\text{Borrower.lastName = BookAuthor.authorName}} \text{ Borrower X BookAuthor}$$

where BookAuthor has (say) 10,000 tuples and Borrower (say) 2000. The join is a cartesian join (and in any case the two relations have no attributes in common so natural join would do the same thing). Therefore, the join will result in generating 10,000 * 2000 = 20 million tuples, each of which might possibly be a part of the results of the query.

However, this join can be computed in ways that differ in cost by orders of magnitude:

1. The simplest scheme would be

```
for (int i = 0; i < 2000; i ++)
{
      retrieve Borrower[i];
      for (int j = 0; j < 10000; j ++)
      {
        retrieve BookAuthor[j];
        if (Borrower[i].lastName ==
            BookAuthor[j].authorName)
          construct tuple from Borrower[i] &
            BookAuthor[j];
      }
}
```

PROJECT

a) This scheme is called NESTED LOOP JOIN.

b) If we assume that each retrieval is accomplished by a separate access to the disk, this would require 2000 disk accesses to read the Borrower tuples plus 20,000,000 disk accesses to read the Borrower tuples, plus however many are needed to write the resulting join - for a total of at least 20,002,000 disk accesses. If each disk access takes 10 ms, this amounts to over 200,000 seconds = 3333 minutes = 55 hours

2. Typically, though, the two relations are physically stored in blocks on disk containing several tuples. Let's assume that each block contains 20 tuples. Then we would only need to access the disk once for each group of 20 tuples. A MUCH BETTER scheme would be:

```
for (int i = 0; i < 2000; i += 20)
{
     retrieve block containing Borrower[i]..Borrower[i+19];
     for (int j = 0; j < 10000; j += 20)
     {
        retrieve block containing BookAuthor[j] ..
                                  BookAuthor[j+19];
        for (int k = 0; k < 19; k ++)
          for (int l = 0; l < 20; l ++)
             if (Borrower[i+k].lastName ==
                       BookAuthor.[j+l].authorName)
                 construct tuple from Borrower[i+k] &
                       BookAuthor[j+l];
     }
}
```

PROJECT

a) At first glance, this looks like a much worse strategy - since we now have 4 nested loops rather than just two.

b) However, when the relative cost of in-memory processing and disk accesses is taken into consideration, this strategy turns out to be much better.

(1)For the outer loop, this would require 100 accesses to retrieve all the Borrower tuples.

(2)The inner loop requires only 500 accesses each time through the BookAuthor table, and this done only 100 times - for a total of 100*500 = 50,000 accesses

(3)The grand total is thus 50,100, plus writes needed for the new relation. This is almost 400 times better than the nested loop join. We're now down to 50,100 * 10ms = 501 seconds (less than 10 minutes).

This strategy is called NESTED BLOCK JOIN.

3. However, if a moderate amount of internal memory is available for buffering, we could do even better. Note that the 2000 Borrower tuples could be stored using just 100 buffers (probably under a megabyte of RAM). We thus consider the following approach:

```
for (int i = 0; i < 2000; i += 20)
     retrieve and buffer block containing
        Borrower[i]..Borrower[i+19];
for (int j = 0; j < 10000; j += 20)
{
     retrieve block containing BookAuthor[j] ..
                                 BookAuthor[j+19];
     for (int k = 0; k < 2000; k ++)
        for (int l = 0; l < 20; l ++)
          if (Borrower[k].lastName ==
                 BookAuthor.[j+l].authorName)
              construct tuple from Borrower[k] &
                 BookAuthor[j+l];
}
```

PROJECT

Now we only need 100 + 500 = 600 disk accesses for reading each of the two relations exactly once! Further, it is clear that this is the best we can do, since we must consider each tuple of each relation at least once, and the two relations together occupy 600 blocks. Our time is now down to about 600*10 ms = 6 seconds!

C. Natural joins (or theta joins based on the equality of some attribute values) can be greatly expedited if indices are available

1. In computing a join, we can scan through the tuples of one relation. For each tuple, we find the tuple(s) of the other relation that join with it (if any) and construct a new tuple for each one found.

2. In the worst case, finding matching tuples in the second relation would require a sequential scan of that relation. Note that this means that we would have to read through the second relation one complete time FOR EACH TUPLE (OR MORE LIKELY EACH BLOCK) in the first relation.

3. However, if the second relation has an index or indices on the join field(s) (or even on one of the join fields if there is more than one), then the sequential scan of the second relation can be avoided. Instead, we use the index to locate tuples in the second relation that are candidates for joining with the current tuple in the first relation.

   Example: suppose we are computing Borrower |X| CheckedOut Suppose that there are 2000 Borrower tuples and 1000 CheckedOut tuples. Suppose further that records are blocked 20 per block (so Borrower is 100 blocks and CheckedOut is 50) Suppose we cannot buffer either table entirely, but can buffer a block from each table. (While buffering an entire table might be feasible for the sizes presented here, it would not be for larger tables, for which the issues are the same.)

   a) In the absence of indices, we can compute this join in one of two ways:

(1)Scan the CheckedOut tuples. For each one, scan the Borrower tuples, looking for matches on the join field (borrowerID). If we use nested block join, we can do this processing in 50 + 50 * 100 = 5050 disk accesses.

(2)Scan the Borrower tuples. For each one, scan the CheckedOut tuples, looking for matches on the join field (borrowerID). This turns out to require a 5100 disk accesses for the nested block join - which is slightly worse.

b) However, suppose Borrower is indexed on the borrowerID attribute. We can now process the join as follows:

Scan the 1000 CheckedOut tuples. For each one, use the borrowerID index on Borrower to locate the matching tuple. (There will be exactly one match for each.) This entails processing only 2000 tuples in all (1000 of each) - an apparent factor of 100 improvement over either prior strategy!

Note, however, that the apparent improvement is not quite what it seems to be, for two reasons:

(1)It is possible that each Borrower tuple will require a separate disk access, since there is no guarantee that two successive CheckedOut tuples will match borrowers in the same block. In fact, in the worst case we could need 50 + 1000 = 1050 disk accesses, which is only about 5 times better (but still worth doing!) Of course, if we can buffer part of the Borrower table, some of our accesses will be to blocks already in the buffer, reducing the number of accesses. (And the more buffers, the higher the probability of a “hit” to a block already in memory.)

(2)Using the index adds additional overhead - especially if the index is stored as a BTree with height greater than 2, and we can't buffer all of level 2. (We assume we can buffer the root of the BTree.) For example, if looking up each borrower takes 2 disk accesses - one to the index and one for the data - we would need a total of 2050 accesses. Now the improvement is only about 2.5 - but still worthwhile.

D. Because an appropriate index can greatly speed a natural join - especially if it can be buffered in memory rather than stored on disk - it may be desirable in some cases for the query processor to create a TEMPORARY INDEX for one of the relations being joined - to be discarded when the query has been processed.

Example: Suppose neither Borrower nor CheckedOut had an index. If we had to do the join Borrower |X| CheckedOut, we might choose to build a temporary index for one of the tables - say Borrower. (This is preferable, since we expect each CheckedOut tuple to join with exactly one Borrower tuple, but we don't expect each Borrower tuple to actually participate in a join at all, and some may join with several CheckedOut tuples)

1. Each entry in the index for Borrower might occupy on the order of 10 bytes. If we use a dense index (as we must unless Borrower is physically ordered by borrowerID) then the overall index will be about 20,000 bytes long - not a problem for main memory on a machine of any size.

2. Constructing the index, then, will require processing each of the tuples of Borrower once - for a total of 2000 tuples or 100 disk accesses

3. The join itself would require 1050 accesses, as calculated above - for a total of 1150 accesses in all - a worthwhile improvement over not using an index.

E. Natural joins can also take advantage of the PHYSICAL ORDER of data in the database. In particular, if two relations being joined are both physically stored in ascending order of the join key, then a technique known as MERGE JOIN becomes possible:

1. The following is the basic algorithm:

```
get first tuple from Borrower;
get first tuple from CheckedOut
while (we still have valid tuples from both relations)
{
     if (Borrower.borrowerID == CheckedOut.borrowerID)
     {
       output one tuple to the result;
       get next tuple from CheckedOut
       // We might have more checkouts for this borrower,
       // so keep current borrower tuple
     }
     else if (Borrower.borrowerID < CheckedOut.borrowerID)
       get next tuple from Borrower;
     else
       get next tuple from CheckedOut;
}
```

PROJECT

2. Using this strategy, we only fetch each tuple once, for a total of 2000 + 1000 tuples, or 150 disk accesses!

3. Note that the efficiency we attain here is the same as what we would get if the two relations were physically clustered together by borrowerID; however, here the only requirement is that they both be physically ordered by borrowerID, not that they be stored in the same cluster.

F. The book discusses another strategy called HASH JOIN which can be used with hash indices in which we use the hashing function(s) to identify sets of blocks that can contain the same values of the join attributes.

G. One other factor we need to consider when doing two or more joins is join order. When there are multiple joins involved, performance may be very sensitive to the order in which we do the joins.

Example: suppose we want to print out a list of borrowers together with the authors of books they have out. This would involve a query like:

$$\pi_{\substack{\text{lastName,}\\ \text{firstName,}\\ \text{authorName}}} \quad \text{Borrower |X| BookAuthor |X| CheckedOut}$$

1. Suppose, for now, that there are 2000 Borrowers, 1000 CheckedOuts, and 10,000 BookAuthor tuples. Suppose, further, that each Book has an average of two authors (so we expect each CheckedOut tuple to join with two BookAuthor tuples).

2. Natural join is a binary operation, so the three-way join would normally be done by joining two tables, then joining the result with the third. Since natural join is both associative and commutative, this means that there are basically three ways to perform our joins:

   (Borrower |X| BookAuthor) |X| CheckedOut
   (BookAuthor |X| CheckedOut) |X| Borrower
   (Borrower |X| CheckedOut) |X| BookAuthor

   (each of these has commutative variants which have no effect on the actual work involved in performing the operation, so we ignore these variants).

3. In each case, we create a temporary table by joining two tables, then join this with the third. What is interesting is to consider the size of the temporary table created by each order.

   a) In the first case, we join two tables that have no attributes in common, so the natural join is equivalent to a cartesian join. The temporary table has 20 million tuples!

   b) In the second case, since each book has, on the average, 2 authors, we expect the temporary table to contain 2 x 1000 = 2000 tuples.

   c) In the third case, since each CheckedOut tuple is paired with exactly one Borrower, we expect the temporary table to contain 1000 tuples.

Clearly, one of these join orders is best, one is nearly as good, and one is really bad.

4. We will look at formalizing the reasoning we have done here by using statistical data about the database tables later in this lecture. For now we note that the amount of work needed to satisfy a query can be very sensitive to join order.

   a) Simple DBMSs may simply perform joins in the order implied by the code.

   b) Good DBMSs may rearrange joins in order to minimize the size of temporary table(s).

IV. Rules of Equivalence For Queries

A. The first step in processing a query is to convert it from the form input by the user into an internal form that the DBMS can process. This step is called query parsing.

1. This task is not different in principle from the parsing done by a compiler for a programming language, so we won't discuss it here.

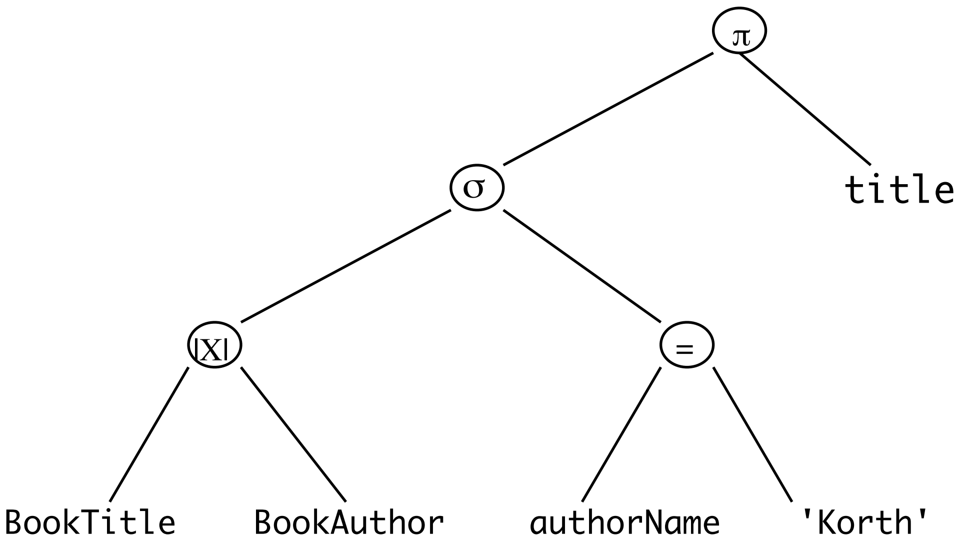
2. The internal form may well be some sort of tree - e.g. our first example (titles of books written by Korth), if formulated as

```
select title
   from BookTitle natural join BookAuthor
   where authorName = 'Korth'
```

is equivalent to the relational algebra expression

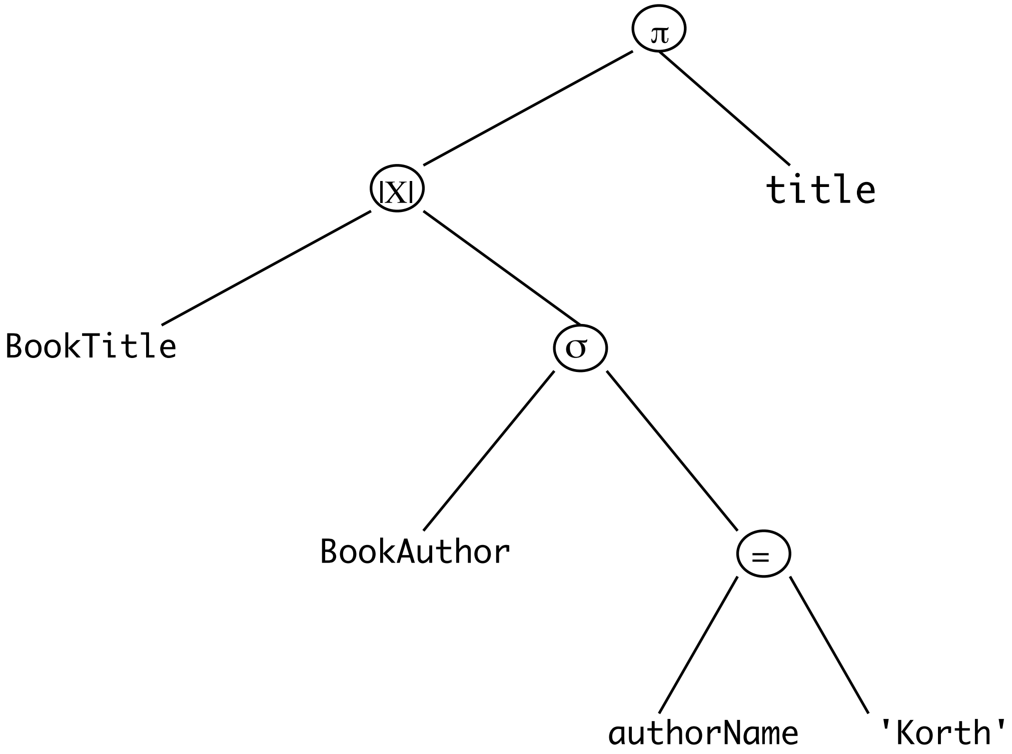
$$\pi_{\text{title}} \ \sigma_{\text{authorName = 'Korth'}} \ \text{BookTitle |X| BookAuthor}$$

which in turn is equivalent to the following tree



3. However, the relational algebra expression could be transformed into a second, equivalent query by an operation on the tree (moving selection inside join)

π_{title} BookTitle |X| $\sigma_{\text{authorName = 'Korth'}}$ BookAuthor



4. However, for our purposes it will suffice to proceed as if relational algebra were the internal form, since a tree like this can always be uniquely constructed from a given relational algebra query.

B. In general, a good query processor will develop a number of different strategies for a given query, then choose the one that appears to have the lowest overall cost.

1. We say that two formulations of a query are EQUIVALENT if they produce the same final answer, except for a possibly different order of the rows (relations are sets, so order is not important.) [Unless, of course, the query includes an "ordered by" term.] Thus, both of our formulations in the previous example were equivalent.

2. The usual cost measure for a query is total disk accesses This is because the cost of a disk access is so high relative to other operations. In general, we will prefer the strategy that processes one of the equivalent forms of the original query with the fewest disk accesses.

3. Obviously, the number of alternatives explored will vary both with the sophistication of the query processor and the size of the query. For simple queries, in fact, it may be the only one alternative will be available.

   Example: $\pi_{\text{callNo}} \quad \sigma_{\text{authorName = 'KORTH'}} \quad \text{BookAuthor}$

   This query can only be processed in one way, since it only involves two operations, and the projection of callNo cannot be done until after the right tuples have been selected using the authorName attribute.

C. Query optimizers generate equivalent formulations of a query to consider by using various rules of equivalence.

1. The book gives a number of these rules, which we can look at briefly.

   PROJECT POWERPOINTS

2. It turns out that there are some transformations that are almost always beneficial:

   a) Do selection as early as possible (move selection inward).

   (1)Example: if we have

   $\sigma_{\text{SomeExpression}}$ RelationA |X| RelationB

   and SomeExpression involves only attributes from one of the two relations (say RelationB), then we can convert the query to an equivalent - and usually more efficient - form:

   RelationA |X| $\sigma_{\text{SomeExpression}}$ RelationB

   (This is, in fact, the transformation that was used in the above example)

   (2)Suppose, however, we have a selection expression which involves attributes from BOTH relations in the join. In this case, it may not be possible to move the selection operation inward.

   Example: we considered the following query earlier:

   $\sigma_{\text{Borrower.lastName = BookAuthor.authorName}}$ Borrower X BookAuthor

   Clearly, this requires us to do the join before we can do the selection.

(3)However, sometimes when a selection expression involves attributes from both relations in a join we can still more selection inward by looking at the structure of the selection expression itself.

Example: we might want to find out what books (if any) that cost us more than $100.00 to buy are now overdue. This requires the query:

$$\sigma_{\text{purchasePrice} > 20.00 \text{ and dateDue} < \text{today}} \text{ Book |X| CheckedOut}$$

By taking advantage of the fact that any selection condition of the form:

$$\sigma_{\text{ConditionA and ConditionB}}$$

is equivalent to

$$\sigma_{\text{ConditionA}} \; \sigma_{\text{ConditionB}}$$

our query is equivalent to:

$$\sigma_{\text{purchasePrice} > 20.00} \; \sigma_{\text{dateDue} < \text{today}} \text{ Book |X| CheckedOut}$$

or

$$(\sigma_{\text{purchasePrice} > 20.00} \text{ Book }) \text{ |X| } (\sigma_{\text{dateDue} < \text{today}} \text{ CheckedOut})$$

b) A second heuristic is similar to the first: do projection as early as possible (move projection inward.)

(1)The motivation here is that projection reduces the number of columns in a relation - hence the amount of data that must be moved around between memory and disk, or stored in a temporary relation in memory. In particular, if a query involves constructing an intermediate result relation, then use of this heuristic may result in

(a)being able to keep the temporary relation in memory, rather than storing it on disk

(b)or allowing more tuples of the temporary relation to be stored in one block - thus reducing disk accesses.

(2)Example:

$$\pi_{\substack{\text{lastName,}\\ \text{firstName,}\\ \text{title,}\\ \text{dateDue}}} \text{ Borrower |X| CheckedOut |X| Book}$$

could be done somewhat more efficiently as:

$$\pi_{\substack{\text{lastName}\\ \text{firstName}\\ \text{title}\\ \text{dateDue}}} \text{ Borrower |X| } (\pi_{\substack{\text{borrowerID}\\ \text{title}\\ \text{dateDue}}} \text{ CheckedOut |X| Book})$$

which reduces the size of the temporary table created by the first join. (Note that we need to keep one attribute from this join that we don't need in the final result to allow the second natural join to be done.)

(3)The benefits gained by this heuristic may not be as great as those from the move selection inward heuristic - but it's still worth considering.

V. Use of Statistical Information to Choose an Efficient Query Processing Strategy

A. We have already noted that a DBMS may keep some statistical information about each relation in the database.

1. The following statistics may be kept for each table.

a) For each relation r, the total number of tuples in the relation. We denote this n_r.

b) For reach relation r, the total number of BLOCKS. We denote this by b_r.

c) For each relation r, the size (in bytes) of a tuple. We denote this by l_r

d) For each relation r, the blocking factor (# of tuples per block). We denote this by f_r. Assuming tuples do not span across blocks, his is simply floor(blocksize / l_r)

(Actually, if we can compute some of these from the others, so we don't need to keep all of them - e.g. the following relationship holds among the above if the tuples of a relation are stored in a single file without being clustered with other relations.

b_r = ceiling(n_r / f_r)

2. The following statistics may be kept for each column of a table

a) For each attribute A of each relation r, the number of different values that appear for A in the relation. We denote this V(A,r).

Note that if A is a superkey for r, then V(A,r) = n_r

b) A corollary of this is the fact that, for any given value that actually occurs in r, if A is not a superkey then we can estimate the number of times the value occurs as:

n_r / V(A, r)

(where this is just an estimate, of course - the true count for a given value could be as small as 1, or as many as n_r - V(A,r) + 1.)

c) It may also be desirable, in some cases, to store a histogram of the relative frequency of values lying in different ranges.

3. Fortunately, the table statistics are very easy to maintain - in fact, they're needed in the meta-data in any case. The column statistic is harder to maintain in general, but V(A,r) is relatively easy if the attribute A is indexed (just maintain a count of index entries). Fortunately, V(A,r) tends to be of most interest for those attributes A which also tend to be prime candidates for indices.

B. We can use these statistics to estimate the size of a join.

1. In the case of the cartesian product r X s, the number of tuples is simply n_r * n_s, and the size of each tuple in the result is $l_r + l_s$.

2. In the case of a natural join r |X| s, where r and s have some attribute A in common, we can estimate the size of the join two ways:

a) Estimate that each of the n_s tuples of s will join with

n_r / V(A, r) tuples of r

b) Estimate that each of the n_r tuples of r will join with

n_s / V(A, s) tuples of s

c) The first formulation is equivalent to n_r * n_s / V(A, r) and the second is equivalent to n_r * n_s / V(A, s). Clearly, these are two different numbers if V(A,r) ≠ V(A, s). However, if this is the case, then it must be that some of the values of A that occur in one table don't occur in the other - so we want to use the smaller of the two estimates - leading to the following estimate for the size of r |X| s:

min(n_r * n_s / V(A,r), n_r * n_s / V(A, s)) =
n_r * n_s / max(V(A,r), V(A,s))

d) Of course, this estimate could be far from correct in a particular case.

Example: suppose we performed a natural join between tables for CSMajorsAtGordon and PsychologyMajorsAtGordon, based on studentID.

Since V(id, CSMajorsAtGordon) = $n_{\text{CSMajorsAtGordon}}$ and
V(id, PsychologyMajorsAtGordon) = $n_{\text{PsychologyMajorsAtGordon}}$

and since $n_{\text{PsychologyMajorsAtGordon}}$ > $n_{\text{CSMajorsAtGordon}}$, we would estimate the size of the join to be $n_{\text{CSMajorsAtGordon}}$ - but it's actually 2 (when I consulted the data while revising this lecture).

However, as a tool for selecting query strategies, these estimates are still very useful - since the alternative of actually carrying out the various strategies and then comparing the costs is hardly helpful!

e) In order to make further estimations, it is also helpful to note that we can estimate V(A, r |X| s) = min(V(A, r), V(A, s)) - i.e. some tuples in the relation having the larger number of values don't join with any tuples in the other relation, and thus don't appear in the result.

C. We now consider how these statistics may be used to help us decide on the order in which to perform multiple joins

1. Earlier, we considered a a query that prints borrower names an authors of books they have checked out.

   π   Borrower |X| BookAuthor |X| CheckedOut
    lastName,
    firstName,
    authorName

2. We saw that the total amount of effort in processing the query varied greatly depending on the order in which the joins are performed. We established this by using informal reasoning to assess the various strategies. We now want to see how statistical data could be used to arrive at the same conclusions algorithmically. To review, the join orders we want to compare are:

   (Borrower |X| BookAuthor) |X| CheckedOut
   (BookAuthor |X| CheckedOut) |X| Borrower
   (Borrower |X| CheckedOut) |X| BookAuthor

3. Suppose the relevant statistics have the following values (all recorded in the meta-data or calculated form the meta-data):

   a) $n_{\text{Borrower}} = 2000$

   b) $n_{\text{CheckedOut}} = 1000$

   c) $n_{\text{BookAuthor}} = 10000$

   (these are the values we used in the example)

d) V(borrowerID, Borrower) = 2000 (since borrowerID is a key for Borrower, each tuple must have a distinct value)

e) V(borrowerID, CheckedOut) = 100 - so we expect each borrowerID that occurs at all to occur in 1000/100 = 10 CheckedOut tuples

f) V(callNo, CheckedOut) = 500 - so we expect each callNo that occurs at all to occur in 1000/500 = 2 CheckedOut tuples

g) V(callNo, BookAuthor) = 5000 - so we expect each callNo to occur in 10000/5000 = 2 BookAuthor tuples

4. Let's now consider how many intermediate result tuples we would expect each join order to produce:

a) (Borrower |X| BookAuthor) |X| CheckedOut

Temporary table needed for Borrower |X| BookAuthor - no attributes in common (cartesian join)

estimated $n_{\text{Borrower |X| BookAuthor}} = n_{\text{Borrower}} * n_{\text{BookAuthor}} =$ 2000 * 10,000 = 20 million

b) (BookAuthor |X| CheckedOut) |X| Borrower

Temporary table needed for BookAuthor |X| CheckedOut - join attribute = callNo

$n_{\text{BookAuthor}} = 10{,}000$; $n_{\text{CheckedOut}} = 1000$,
V(callNo, BookAuthor) = 5000; V(callNo, CheckedOut) = 500

estimated $n_{\text{BookAuthor |X| CheckedOut}} = \min(10{,}000 * 1000 / 5000,$ 10,000 * 1000 / 500) = min(2000, 20,000) = 2000

c) (Borrower |X| CheckedOut) |X| BookAuthor

Temporary table needed for Borrower |X| CheckedOut - join attribute = borrowerID

n_{Borrower} = 2000; $n_{\text{CheckedOut}}$ = 1000,
V(borrowerID, Borrower) = 2000; V(borrowerID, CheckedOut) = 100

estimated $n_{\text{Borrower |X| CheckedOut}}$ = min(2000*1000 / 2000, 2000*1000 / 100) =1000