Module 6: Distributed Query Optimization

In the previous modules, we transformed SQL queries into relational algebra expressions, and then transformed them into query trees. We mapped them onto relation fragments using techniques called *decomposition* and *data localization*. We used query tree transformation rules into more optimized expressions. However, we did not address optimizing query joins, which are the most expensive part of query statements. There are many ways to *order* and *execute* the join statements. A *query optimizer* attempts to find the best join ordering and execution strategy.

Finding the best join ordering is exponential with respect to the number of relations that need to be ordered. It is too costly to find the *best* ordering in real time. Rather, the purpose of the optimizer is to find good join ordering in a reasonable amount of time. We use heuristics to approximate the optimal performance. The result of applying heuristics is a query execution plan.

Execution Plan Components

It is impossible to predict the actual execution time of a query a priori. However, we can approximate execution based on cost estimates of disk I/O, CPU time, and network communication time. It used to be that communication costs were dominant, but in many circumstances, I/O and CPU costs are on the same order of magnitude. Therefore, they need to be accounted for as well.

Cost-Based Query Optimization

Much of query optimization is independent of *centralized* versus *distributed* database systems. Since the input is just a set of relational algebra statements, the tables can be whole, as in the case of both centralized and distributed systems, or fragmented tables, as in the case of distributed systems. Query optimization requires the formulation of an objective function that characterizes costs of queries. Ultimately, the query optimizer takes as input a query expressed in relational algebra and produces a *query execution plan (QEP)* based on selecting the plan with the minimal cost.

Query Optimization Steps

(Watch Module 6A video – Query Optimization Steps)

Cost Functions

Cost functions generally either estimate the *total* time or *response* (or execution) time of a query. While it may be tempting to choose the query plan with the best estimated response time, it may not be the best choice for the system in general. It is possible that the query with the best estimated response time ties up resources that other queries could use, thus reducing the overall throughput of queries through the system. Rather, selecting the query plan that minimizes costs of system resources will generally improve the throughput of the system.

We introduce a simplified cost function that is measured in terms of total time to execute a query, which is a function of CPU time, I/O time, time to instantiate a connection between distributed platforms, and time to pass messages between platforms:

$$T_{total} = T_{CPU}*\#inst + T_{I/O}*\#I/O + T_{MSG}*\#msgs + T_{TR}*\#bytes$$

where:

T_{total} = total estimate time used by system resources execute the query

T_{CPU} = average CPU time of database operations
#inst = number of database operation instructions

$T_{I/O}$ = average time of a database I/O operation
#I/O = number input and output operations

T_{MSG} = average time to instantiate a network message connection
#msgs = number network messages

T_{TR} = average time to transfer a byte of data over the network
#bytes = number bytes transferred over the network

Of course, none of the T_x measurements are actually constant. T_{TR} depends on network characteristics at a given instant. T_{CPU} and $T_{I/O}$ depend on local demand placed on the CPU or I/O streams by other queries, programs, or other system operations.

Communication time is defined as:

$$CT(\#bytes) = T_{MSG}+T_{TR}*\#bytes$$

Communication time is linear with respect to the number of bytes transferred. This is not really constant. It depends on factor such as network traffic, distance between sites, and data repeater characteristics. On a wide area network (WAN), communication time usually dominates; on a local area network (LAN), there is a balance among components of the cost model.

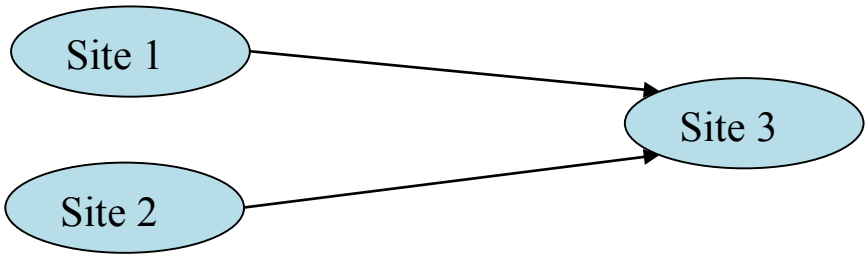
As stated previously, it is preferable to estimate costs based on resources used rather than response time. This is because throughput is a better metric to optimize than response time because it is a better indicator of whether all the queries are optimized rather than just one query. However, here is a modified cost function that optimizes response time.

$$T_{resp} = T_{CPU}*\#par\_inst + T_{I/O}*\#par\_I/O + T_{MSG}*\#par\_msgs + T_{TR}*\#par\_bytes$$

where:

#par_inst = number of parallel database operation instructions
#par_I/O = number of parallel input and output operations
#par_msgs = number of parallel network messages
#par_bytes = number of parallel bytes transferred over the network

To illustrate, suppose *x* units of data are to be sent from Site 1 to Site 3 while *y* units of data are to be sent from Site 2 to Site 3.



The estimated communication cost of all resources used during the transfers is:

$T_{total} = T_{MSG} +$ // time to instantiate communication between 1 and 3
$T_{MSG} +$ // time to instantiate communication between 2 and 3
$T_{TR} * x +$ // time to transfer x bytes from 1 to 3
$T_{TR} * y$ // time to transfer y bytes from 2 to 3

$= 2*T_{MSG}+T_{TR}*(x+y)$

while, if all the operations are scheduled in parallel, the estimated response time is:

$T_{resp} = \max (T_{MSG}+T_{TR}*x, T_{MSG}+T_{TR}*y) = T_{MSG}+T_{TR}*(\max (x,y))$

Database Statistics

A main factor in estimating performance of a query is estimating the size of intermediate relations and the cost to transmit them over the network. Since network communication is a dominant cost in distributed query processing, it is desirable to minimize the size of intermediate relations. The sizes of intermediate relations can be estimated by examining statistics of stored relations.

There is a trade-off between the precision and cost of statistics. Better statistics yield more accurate cost estimates. However, more time must be spent accumulating and aggregating them. More disk space must be allocated to store rich statistics as well. Less accurate statistics are less costly to store and require less time to accumulate. However, less accurate statistics may also lead to less accurate cost estimates, which may lead to slower queries overall.

Query Optimization Statistics

(Watch Module 6B video – Query Optimization Statistics)

The following serve as a reference for the concepts discussed in the video:

Basic Statistics

- Assume attributes A = { A_1, …, A_n } for relation R fragmented into R_1, …, R_r
    - *length*(A_i) – length in bytes of A_i in each R_j
    - *card*($\Pi_{Ai}(R_j)$) – # actual distinct values of A_i in R_j
    - *min*(A_i) and *max*(A_i) - min and max values of domain of A_i
    - *card*(*dom*[A_i]) – # of potential unique values of A_i in domain
    - *card*(R_j) – # of tuples in R_j

- *Join selectivity factor*
    - SF_J = (card(R ⋈ S) / (card(R) * card(S))
- Size of relation
    - *size*(R) = card(R) * length(R)

Cardinality Estimates

- Simplification assumptions - not always true!
    - Attribute values are *uniformly distributed*
        - For example, assume that a customer table has a column *customerAge*. We assume for query optimization purposes that the values of *customerAge* are uniformly distributed such that we expect an equal number of 20 year olds, 40 year olds, and 100 year olds in the table. In truth, this assumption is almost always false. However, it is a useful assumption for query optimization.
    - Attributes are *independent*
        - For example, assume that an employee table has columns *jobTitle* and *salary*. We assume for query optimization purposes that they are independent, even though it is likely that one's job title corresponds roughly to one's salary.

- Selection
    - card (σ_F(R)) = $SF_{S(election)}$(F) * card(R)
        - F is dependent of formula
    - SF_S(A = value) = 1/ card(Π_A(R))
        - If the range of *customerAge* is 21-100, then the probability of being any specific age is 1/(100-20) = 1/80
    - SF_S(A > value) = (max(A) - value) / (max(A) - min(A))
        - If the range of *customerAge* is 21-100, the probability of an age over 60 (not including 60) is (100-60)/(100-20) = 40/80 = 1/2
    - SF_S(A < value) = (value - min(A)) / (max(A) - min(A))
        - If the range of *customerAge* is 21-100, the probability of an age under 31 (not including 31) is (30-20)/(100-20) = 10/80 = 1/8
    - $SF_S(p(A_i) \wedge p(A_j)) = SF_S(p(A_i)) * SF_S(p(A_j))$

        - p(A) is predicate of A
        - Notice that A_i and A_j are different columns! This does not work if they are the same column.
        - If the probability of someone being over 60 is ½ and the probability of someone having brown hair is ¼, then the joint probability of both being over 60 and having brown hair is ½ * ¼ = 1/8.
    - $SF_S(p(A_i) \vee p(A_j)) = SF_S(p(A_i)) + SF_S(p(A_j)) - (SF_S(p(A_i)) * SF_S(p(A_j)))$
        - Notice that A_i and A_j are different columns! This does not work if they are the same column.
        - If the probability of someone being over 60 is ½ and the probability of someone having brown hair is ¼, then the joint probability of both being over 60 or having brown hair is ½ + ¼ - 1/8 (so we don't double count the people with both brown hair and over 60) = 5/8.
    - $SF_S(A \in \{values\}) = SF_S(A = value) * card(\{values\})$

- Projection
    - For primary key value A: $card(\Pi_A (R)) = card(R)$

- Cartesian Product
    - card (R X S) = card(R) * card(S)

- Equijoin
    - $card(R \bowtie_{A=B} S) = card(S)$
    - primary key in R and foreign key in S

- Other joins
    - $card(R \bowtie S) = SF_J * card(S) * card(R)$
    - Unfortunately, SF_J is defined in terms of $R \bowtie S$

Query Processing of Joins

We will examine the two most common approaches to query processing of joins. The first is Ingres, which was invented by Mike Stonebreaker et al. at Berkeley in the 1970's. The second is System R, which was invented by Astrahan et al. and Selinger et al. at IBM San Jose Research Labs in the 1970s. Most query join processing algorithms are variations of either the Ingres or System R algorithms.

Ingres

Ingres is a dynamic query optimization algorithm because it executes queries while it performs query planning and optimization. It breaks apart a query into a series of table joins that can be represented as linear trees. It attempts and attempts to order the table joins to minimize the number of intermediate results at each step of the query.

Ingres Algorithm

(Watch Module 6C video – Ingres Algorithm)

System R

System R is a static query optimization algorithm because it first determines an optimal query plan and then executes it. Ideally, one would create an exhaustive list of the possible query plans and estimate which one would minimize the query costs. Unfortunately, this is not feasible because of the number of possible query plans is exponential with respect to the number of tables. Rather, System R identifies a cost model and then applies heuristics to estimate which query plan is likely to perform best.

The query optimization heuristics build a tree in which each top level branch represents the beginning of a query plan. Specifically, the cost model emphasizes selecting the most cost efficient join ordering. It prunes those branches that are estimated to be expensive with respect to the cost model. With the remaining branches, it builds out the next level of possible query plans, specifically focused on join ordering. It applies the cost model again to prune those branches that are estimated to be expensive with respect to the cost model. The algorithm continues in this respect until one plan is identified as the least cost plan.

The cost model is based on statistics that help determine the size of both initial tables and the product of intermediate joins as well as selectivity factors. The following video illustrates this concept.

System R Algorithm

(Watch Module 6D video – System R Algorithm)

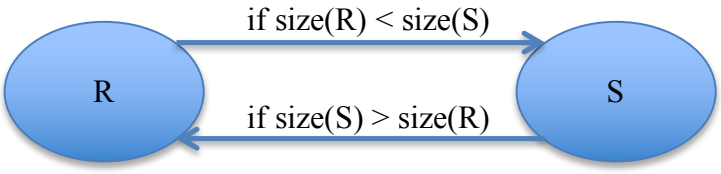
Query Optimization on Distributed Databases

Both INGRES and System R have corresponding algorithms for distributed query processing. These are known as Distributed Ingres and R*, respectively. They both use regular joins, not semi-joins.

In the subsequent discussion, there is no need to differentiate between relations and fragments; they are treated identically. Also, local processing times are ignored. We examine only joins that have relations at different sites.

Join Ordering for Two Tables

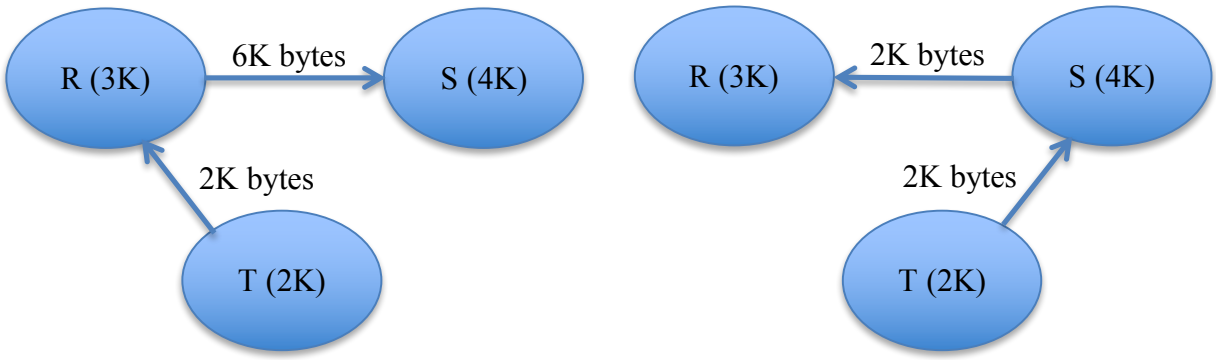
Take for example R ⋈ S. Our choices are to move the contents of R to the site where S is stored or to move the contents of S to the site where R is stored. We will perform the join at the target site.



In this case, the choice appears to be simple. With nothing else to consider, the option that moves the least amount of data across the network appears to be the best. This is simple when considering two relations.

Join Ordering for More than Two Tables

For optimizing join order for more than two tables, it is important to understand the size of the data for intermediate results because intermediate results must also be transferred over the network. Database statistics are necessary to estimate these sizes accurately. For example, let us assume that we wish to join the following tables: R – EMP; T – PROJ; S – ASG. Two possible join plans are:



The first plan moves T to R, performs the cross product R X T, moves the intermediate result to S, and performs S ⋈ (R X T). The second plan moves T to S, performs the join S ⋈ T, moves the intermediate result to R, and performs R ⋈ (S ⋈ T). These plans can be evaluated if the following estimates are know: size(R), size(S), size(T), size (R X T), size(S ⋈ T). Based on the intermediate join sizes, the second plan appears to be better because only 4K bytes are moved across the network as opposed to the first plan's 8K bytes.

The following query plans must also be evaluated to identify the estimated least expensive query plan:

- R -> T; R X T -> S
- T -> R; R X T -> S
- S -> T; S ⋈ T -> R
- T -> S; S ⋈ T -> R
- R->T, S->T; R ⋈ (T ⋈ S)
- R->T, S->T; (R X T) ⋈ S

If a good estimate of the intermediate sizes of tables are not known:

- Distributed INGRES picks the product of cardinality – this estimate is very high because it only occurs for a cross product

- System R* picks a fraction of product of cardinality – this is a more reasonable estimate

Semi-Joins

Semi-joins were invented by Bernstein and Chiu in 1981 to address large communications costs incurred by sending full tables from one site to another. The following example illustrates the semi-join algorithm. Suppose table A is located on site 1, with attributes a_1, a_2, …, a_n, and table B is located on site 2 with attributes b_1, b_2, …, b_m. Furthermore, consider the following SELECT statement:

SELECT a_3, a_6, a_7, b_2, b_5, b_8
FROM A, B
WHERE $a_3 = b_5$.

A simple way to solve the query is to send table A to site 2 to be joined with B. Alternatively, table B can be sent to site 1 to be joined with A. In either case, an entire table is sent to the other site, which may incur high communication costs. An alternate algorithm that will join tables A and B and potentially reduce the communication costs is the semi-join algorithm.

The semi-join algorithm uses the partial $\ltimes$ (bowtie) symbol. Its results are equivalent to an equi-join: $A \bowtie B = A \bowtie (B \ltimes A)$

1. Select (DISTINCT) the join column(s) of A. In this example, the join column is a_3.
   $A' \leftarrow \prod_{a3}(A)$
2. Send A' to site 2
3. Join A' with B and select the columns of B from the example query.
   $B' \leftarrow \prod_{a3, b2, b5, b8} (A' {}_{a3} \bowtie {}_{b5} B)$
   B' is now $= (B \ltimes A)$
4. Send B' to site 1
5. Result $= \prod_{a3, a6, a7, b2, b5, b8} (A {}_{a3} \bowtie {}_{b5} B')$

Of course, A and B could be reversed in this example to solve the query via the equivalence $A \bowtie B = (A \ltimes B) \bowtie B$. In fact, the best strategy is determined by assigning costs to each operation of $A \bowtie B$, $A \bowtie (B \ltimes A)$, and $(A \ltimes B) \bowtie B$ and finding the minimum overall cost.

Example Cost-Based Analysis

Let us evaluate the cost joining DEPARTMENT $\bowtie$ CLASS where there is a 1 to many relationship between DEPARTMENT and CLASS.

- Table DEPARTMENT has a primary index on attribute DeptID
- There are 50 Departments in the University
- Table DEPARTMENT has the following fields
  - DeptID – 4 bytes
  - DeptName – 20 bytes
  - DeptHead – 40 bytes

    - DeptSize – 2 bytes (values between and including 1 to 500)
- Table CLASS has an index on the foreign key DeptID
- There are 1000 Classes in the University
- Table CLASS has the following fields
    - DeptID – 4 bytes
    - ClassID – 4 bytes
    - ClassName – 20 bytes
    - Professor – 40 bytes
    - ClassSize – 2 bytes (values between and including 11 to 50)

Let's evaluate the following query:

```
SELECT    DeptName, ClassName, Professor
FROM      DEPARTMENT, CLASS
WHERE     ClassSize > 40 and DeptSize > 400
```

Let's evaluate the query by performing a semi-join on relations DEPARTMENT and CLASS.

- Cost =

  T_{CPU}* card(DEPARTMENT) = T_{CPU} * 50 rows

  // The first step is to read the rows of DEPARTMENT, which there are 50, and multiply by the cost of the read operation, T_{CPU}.

  //During the read, we will keep only the number of departments of DeptSize > 400. We estimate this to be (500-400)/(500-0) = 1/5. We will transfer a table DEPARTMENT' which contains only the DeptID's for those rows that meet the constraint of DeptSize > 400, which is estimated to be 1/5 of 50 rows = 10 rows.

  + T_{MSG} + T_{TR} * size(DEPARTMENT') = T_{MSG} + T_{TR} * 40 bytes

  //The second step is to calculate the cost of transmitting the rows of DEPARTMENT' to the site where CLASS is stored. The cost of establishing the connection is T_{MSG}. The cost of transmitting DEPARTMENT' is calculated by multiplying the cost of transmitting a byte over the network, T_{TR}, times the number of DeptID bytes of DEPARTMENT'. This is 10 rows x 4 bytes = 40 bytes.

  + T_{CPU}* card(DEPARTMENT') = T_{CPU} * 10 rows

  //The third step is to calculate the cost of writing DEPARTMENT' to the site where CLASS is stored. This cost is the product of the cost of a write operation, T_{CPU}, and the number of rows, card(DEPARTMENT'). In reality, the costs of reads and writes are different, but for simplicity, we are approximating both of them to be T_{CPU}.

+ T_{CPU}* card(DEPARTMENT') + T_{CPU}* card(DEPARTMENT')log(card(CLASS)) +
T_{CPU}* card(CLASS) * SelectivityFactor$_{DeptSize>400}$
= T_{CPU}* 10 + T_{CPU}* 10 * 3 + T_{CPU}* 1000 * 1/5
= T_{CPU} (10 + 30 + 200)
= T_{CPU} * 240

// **For a B+ Tree Index** - The fourth step is to calculate the cost of joining DEPARTMENT' with CLASS. We will make DEPARTMENT' our outer table and that CLASS is our inner table because CLASS has an index defined on it. When writing DEPARTMENT', we did not recreate its index at the new site. Making DEPARTMENT' the outer table means that we read each row of DEPARTMENT', use its key value DeptID' to search an index tree to find a pointer to the corresponding rows of CLASS, and then read the corresponding rows of CLASS. The cost of reading each row of DEPARTMENT' is T_{CPU}* card(DEPARTMENT'). The cost of searching the index tree of CLASS is log(card(CLASS)) as described in a previous module. We must perform the index search for each row of DEPARTMENT', yielding a cost of T_{CPU}* card(DEPARTMENT) log(card(CLASS)). When we find the corresponding row in CLASS, we can follow the rows next to the CLASS row we found to find all rows with that DeptID. The cost of reading each row of CLASS once for each row of DEPARTMENT' is T_{CPU}* card(DEPARTMENT'). Adding the costs yields T_{CPU}* card(DEPARTMENT') + T_{CPU}* card(DEPARTMENT)log(card(CLASS)). Since DEPARTMENT' represents 1/5 of the rows of DEPARTMENT, we expect to match and read about 1/5 of the rows of CLASS, which is 1/5 * card(CLASS) = 1/5 * 1000 = 200. The cost expression is T_{CPU}* card(DEPARTMENT') + T_{CPU}* card(DEPARTMENT')log(card(CLASS)) + T_{CPU}* card(CLASS) * SelectivityFactor$_{DeptSize>400}$ = T_{CPU}* 10 + T_{CPU}* 10 * 3 + T_{CPU}* 1000 * 1/5.

[+ T_{CPU}* card(DEPARTMENT') + T_{CPU}* card(DEPARTMENT')) +
T_{CPU}* card(CLASS) * SelectivityFactor$_{DeptSize>400}$
= T_{CPU}* 10 + T_{CPU}* 10 + T_{CPU}* 1000 * 1/5
= T_{CPU} (10 + 10 + 200)
= T_{CPU} * 220]

//**For a Hashed Index** – Replace all the log(card(CLASS)) with a 1 because the lookup is linear.

+ T_{MSG} + T_{TR}*card(CLASS')*length(columns)
= T_{MSG} + T_{TR} * 50 * 64 bytes
= T_{MSG} + T_{TR} * 3200 bytes

// We read 200 rows from the CLASS table. We also estimate that of those 200 rows, only (50-40)/(50-10) = ¼ of those rows will meet the criteria of ClassSize > 40. Thus, we estimate only 50 rows will be returned. Furthermore, we only will return columns DeptID, ClassName, and Professor, which is 4 + 20 + 40 bytes = 64 bytes. Thus CLASS' has 50 rows and is 64bytes/row.

+ T_{CPU}* card(CLASS')
= T_{CPU}* 50 rows

// We write this into the original site.

// **Important** – Table CLASS' provides the answer to the query. However, if the query required us to return columns from DEPARTMENT as well, we would have to join CLASS' back to DEPARTMENT. We would have to calculate that cost of the join in a similar way that we calculated the join above. We would use CLASS' as the outer table of the join because it has no index. For every row of CLASS', we would use the DeptID value to read the index of DEPARTMENT to find the corresponding row in DEPARTMENT. We would then read the actual row and join the row from CLASS' with the row from DEPARTMENT. Thus, if a join were required, the estimated cost would be:

T_{CPU}*50 rows to read each row from CLASS'
+ T_{CPU} * 50rows * log(card(DEPARTMENT)) to find the corresponding entry in the B+ Tree index [or T_{CPU} * 50 rows * 1 to find the corresponding entry in the Hash index]
+ T_{CPU} * 50rows to read the corresponding rows in DEPARTMENT.

Again, for this specific problem, we do not need to calculate this cost because we do not need any columns from DEPARTMENT.

Thus, adding the costs:

T_{Total} = T_{CPU} * 50 rows
+ T_{MSG} + T_{TR} * 40 bytes
+ T_{CPU} * 10 rows
+ T_{CPU} * 240 [alternative with hashed index: + T_{CPU} * 220]
+ T_{MSG} + T_{TR} * 3200 bytes
+ T_{CPU}* 50 rows

= T_{CPU}*350 + 2*T_{MSG} + T_{TR}*3240
= [alternative with hashed index: T_{CPU}*330 + 2*T_{MSG} + T_{TR}*3240]

Distributed Ingres

(Watch Module 6E video – Distributed Ingres)

R* Algorithm

(Watch Module 6F video – System R* Algorithm)

Conclusion

In this module, we described query optimization techniques for centralized database systems and extended them to distributed database systems. We defined a data

model that represents a query execution plan, which is a set of instructions sent to a query execution processor. We defined algebraic operators that manipulate the data model to ensure that the manipulated model represents the same query, but that the instruction set of the new model is likely to execute faster.

We discussed how to create a search space of all the possible data model structures and we defined a cost model that helps us evaluate which of the possible data models will likely be executed faster. We especially focused on the join because it is one of the most expensive operations in a relational database system. We also developed heuristics to reduce the size of the search space to make selecting the most promising query execution plan a computationally tractable process. Finally, we applied these techniques to a distributed database system.