Module 5: Distributed Query Processing

Introduction to Query Processing

Query processing is a complex process. This module is devoted to describing a step-by-step process of transforming a query statement to query execution plan. A query execution plan is a series of smaller execution steps that together will solve a query. Generally, a query can be solved with multiple query plans, each of which consist of a different set of execution steps executed in a different order. Depending on the specific data in a database, each query plan will likely have a different execution time. Users generally prefer to use the query plan that will run the fastest. One way for a query-processing engine to find the fastest query is to create an exhaustive list of all possible query plans, evaluate them, and select the one that is likely to run the fastest. However, the number of possible query plans that a query processor would have to examine for most queries is a prohibitively large, requiring much time to examine each of the query plans and select the plan that is estimated to complete the fastest. With complicated queries, the process of creating an exhaustive list of query plans, calculating the estimated query times for each plan, and selecting the best option would require users to wait too long until their queries would actually execute. Another way to approach the problem is to apply heuristics that find near optimal query plans in a reasonable amount of time. This approach may not find the very best solution, but will often find in a reasonable amount of time a query that executes reasonably fast. This module will provide both a framework for assigning a cost to each query plan and a method for efficiently searching through the state space of potential query plans to find a near optimal plan in a reasonable amount of time.

An important aspect of query execution planning is the query language. It is the language that must be parsed and transformed into a representation that will be used to describe the query plan. Since the most common relational database query language is SQL, we will focus on it as the language of query expression that we will transform. SQL is non-procedural. The user does not write a program to tell the query processor how to retrieve data. Rather, an SQL expression defines what is to be returned in terms of attributes, constraints, and joins. The *query processor* is responsible for transforming queries into a set of commands that will execute the query and return the results. As previously discussed, each query will have many possible query plans. Some are fast and efficient; others are slow and inefficient. Ultimately, minimizing the cost of query planning plus the query of query processing is crucial to the success of relational database management systems.

Distributed relational database management systems are more complex than centralized relational database systems because there are more factors to consider. In centralized databases, the system often must find optimal joins among multiple tables. In distributed systems, the system must find optimal joins

among multiple fragments. Since the number of fragments is often more than the number of complete tables, the search space among possible query join paths is often much larger. Furthermore, the fragments are distributed which means that a significant portion of the query execution cost to consider is communication time among multiple databases across a network. Moreover, distributed databases often perform replication and managing the replication is more complex. Taken all together, distribution increases both the complexity of query plans and increases the query response time.

Query Processing Example

The purpose of query processing is to execute SQL queries onto a distributed database. The query processor *decomposes* the query to *localized* queries on distributed fragments and pulls their results together. To optimize the performance of the query, the query processor must minimize the costs including I/O, communication, and joins.

The first step of query processing is to translate an SQL query into a relational algebra expression. Then, we will optimize this translation and determine the best query plan. There are many choices of query plans that lead to different efficiencies. Here is an example.

(Watch Module 5A video – Query Processing Example)

The costs that must be considered are the CPU costs for processing queries, I/O to disk, and inter-database communications across the network. Early cost models set communication costs high while CPU and I/O costs were almost negligible. However, with the increase in faster network communications, distributed query cost models now take CPU and I/O costs into consideration. The following discussion describes general database costs.

(Watch Module 5B – Query Operations Costs)

A summary of some of the costs are:

Operation	Cost
Select; Project	• O(n) – on non-index • O(log(n)) – on index • O(1) – on hash index
Project with duplicate elimination; Group By	O(nlog(n)) – for sorting
Join (along index); Semi-join	O(nlog(n))
Join on two sorted tables	O(n)
Cartesian product	$O(n^2)$

Performing Query Optimization

Query optimization can be done by exhaustive search through the state space of possible query plans. However, for any complex query, there are many possible query plans and, as the number of plans increase, the computation of these plans becomes intractable. The other option is to apply heuristics that prune the query plans according to early indicators that lead the query optimizer to believe that certain plans will take too long to execute. While heuristics may not yield the optimal query plan, they will provide reasonable plans in a computationally tractable time.

Another issue is deciding when to perform query optimization. One approach is to perform query planning and optimization for each query the query processor receives. It is expensive to calculate query plans each time. Therefore a strategy would be to compile query plans and reuse them when the same or similar query presents itself again. The cost of query processing is amortized over time. But in a dynamic environment with the data constantly changing, the query optimization algorithms are more accurate if performed for each query. One way to mitigate this is to reuse query plans for some period of time. Then, when it is estimated the plan starts to become stale, recompile.

Factors in Query Optimization

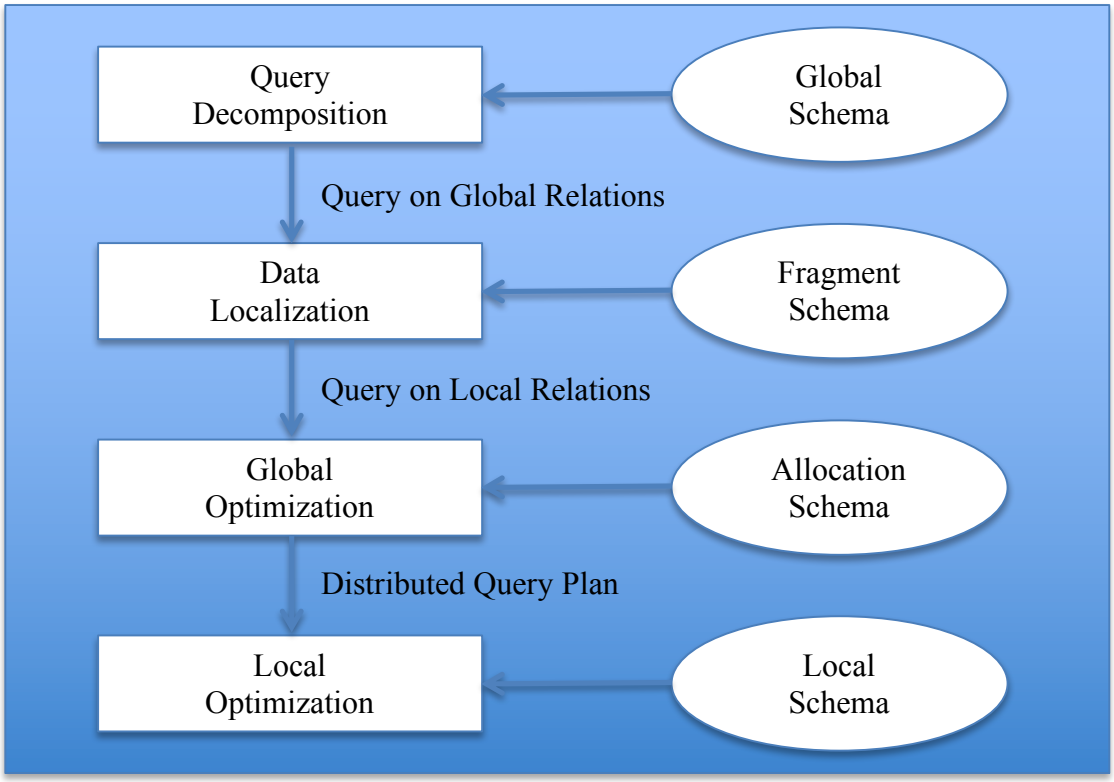
There are several factors that aid in calculating optimal query plans. First, statistics of the database characteristics are critical in determining good query plans. Different statistics might change the query plan. There are trade-offs between static statistics and dynamic statistics. Static statistics do not need to be calculated often, but the numbers go stale. Dynamic statistics require recalculation on a frequent basis, but they are accurate each time query planning is performed. Similarly, there are trade-offs based on how many statistics are collected. A rich set of statistics will make estimation of query costs more accurate, but are more expensive to collect and to store. On the other hand, simpler statistics make query cost estimation less accurate, but are less costly to calculate and store.

Another trade-off exists in deciding where to plan queries. If performed at a central site, it must have knowledge of the entire database. And, all changes to the database that will affect query planning must be maintained at the central site. Alternatively, query planning can take place on distributed sites. It is less expensive to keep the statistics and decisions can be made in parallel.

Knowledge of network topology and statistics is important. Communications costs across distributed systems are critical to query optimization. Replication of fragments is another factor. Efficiencies can be realized through locality of reference of distributed fragments.

Query Processing Layers

The following steps must be done during query processing:



- Query Decomposition
  - *Normalize* the relational algebra query
  - *Analyze* the semantics of the query for correctness
  - *Simplify* the query to eliminate redundancies
  - *Restructure* the query model into a query tree
- Data localization – find the relevant sites
- Global optimization
  - Determine the best ordering of operations
  - Consider I/O time, CPU time, buffer space, communication costs, etc.
  - Select the join ordering that will result in faster query execution times
- Local optimization of queries on individual database sites

Query Decomposition and Data Localization

The following techniques apply equally to centralized and distributed database systems. In the next module, we will address the additional complexities that are particular to query processing in distributed database systems.

Normalization

During query execution, the query processor returns rows from the database tables requested and joined according to the join criteria. Before the rows are returned to the user, the WHERE clause filters out the rows that do not match the query predicates expressed in the WHERE clause. If the WHERE clause predicates are stated in an overly complex manner, the query execution time will be unnecessarily long because of the complex filtering that must be applied to each row. To accelerate query execution, the first step in query processing finds potential simplifications (normalizations) of predicates of the WHERE clause to make constraint checking less complex.

The first step in normalization is to convert the predicate into *conjunctive normal form*. For example, given predicates p1, p2, … p6, the following is a possible WHERE clause constraint. Note that $\vee$ means OR and $\wedge$ means AND.

$$(p1 \vee p2 \vee p3) \wedge (p4 \vee p5) \wedge (p6)$$

The predicates are grouped such that the top level conjunctions are ANDs. *Disjunctive normal form* places the ORs at the top level of the predicate groupings.

$$(p1 \wedge p2 \wedge p3) \vee (p4 \wedge p5) \vee (p6)$$

Conjunctive normal form is more efficient to process because failure to match a row based on any one of the top level predicates means that the row is immediately rejected by the query processor. In contrast, disjunctive normal form is less efficient to process because each top level predicate must be checked before a row is matched. Furthermore, conjunctive normal form allows for certain row matching optimizations based individual predicates. In contrast, disjunctive normal form cannot be optimized on any single predicate because all the predicates must be checked before a row is rejected.

Some Rules for Manipulating Predicates

Given predicates a, b, and c, we can declare the following algebraic laws. These are simple statements of predicate logic.

- Commutative law
  - $a \wedge b \Leftrightarrow b \wedge a$

    - $a \vee b \Leftrightarrow b \vee a$
- Associative law
    - $(a \wedge b) \wedge c \Leftrightarrow a \wedge (b \wedge c)$
    - $(a \vee b) \vee c \Leftrightarrow a \vee (b \vee c)$
- Distributive law
    - $a \wedge (b \vee c) \Leftrightarrow (a \wedge b) \vee (a \wedge c)$
    - $a \vee (b \wedge c) \Leftrightarrow (a \vee b) \wedge (a \vee c)$
- Negation law
    - $\neg(\neg a) \Leftrightarrow a$
- DeMorgan's law
    - $\neg(a \wedge b) \Leftrightarrow \neg a \vee \neg b$
    - $\neg(a \vee b) \Leftrightarrow \neg a \wedge \neg b$

These laws can be applied automatically by a query processor during its normalization phase to simplify following query predicate clause:

```
SELECT    ENAME
FROM      EMP, ASG
WHERE     EMP.ENO = ASG.ENO                AND
          ((RESP = 'Manager'               AND
            DUR = 12)                      OR
           NOT (RESP = 'Programmer'        OR
                 DUR <> 12)
          )
```

The query processor lets a, b, and c represent the following predicates:

a => RESP = ‘Manager’
b => DUR = 12
c => RESP = ‘Programmer’

Then the WHERE clause of the query excluding the join is represented as:

predicate = $((a \wedge b) \vee \neg(c \vee \neg b))$ // Restatement of the constraint
= $((a \wedge b) \vee (\neg c \wedge \neg(\neg b)))$ // DeMorgan’s law
= $((a \wedge b) \vee (\neg c \wedge b))$ // Negation law
= $((a \vee \neg c) \wedge b)$ // Associative law

Converting the predicate clause back to the initial terms, the query processor simplifies the query to:

```
SELECT    ENAME
FROM      EMP, ASG
WHERE     EMP.ENO = ASG.ENO      AND
          ((RESP = 'Manager'     OR
            RESP <> 'Programmer') AND
            DUR == 12)
```

This constraint clause is more efficient to process during query execution than the original constraint clause. By examining this clause, we notice that we could simplify this statement even further by recognizing that

RESP = ‘Manager’ OR RESP <> ‘Programmer’

can be further simplified because the predicate **RESP = ‘Manager’** will always evaluate to *true* when **RESP <> ‘Programmer’**. So, we could further normalize the constraint to:

RESP <> 'Programmer' AND DUR == 12

However, this type of normalization goes beyond the simple algebra normalization laws. It requires knowledge of the semantics of the predicates, which is beyond the capabilities of most query processors.

Decomposition Analysis

Decomposition analysis rejects incorrect types and semantically incorrect queries. An example of type incorrectness is:

```
SELECT    PNAME
FROM      PNO
WHERE     DUR = 'CAD/CAM'
```

The query processor will reject this query because the FROM clause should refer to table names. However, in this case PNO is a column name, which does not make sense from a semantic perspective. In addition, the query processor will reject this query because the WHERE clause predicate should compare DUR to an integer. In this case, DUR is compared to a string, which does not make sense from a type perspective. In general, the query processor will reject queries that do not make sense because of syntax, semantic, and type errors.

Redundancy Elimination

The following laws are used to eliminate redundancies in WHERE clauses:

- $p \wedge p \Leftrightarrow p$
- $p \vee p \Leftrightarrow p$
- $p \wedge true \Leftrightarrow p$
- $p \vee true \Leftrightarrow true$
- $p \wedge false \Leftrightarrow false$
- $p \vee false \Leftrightarrow p$
- $p \wedge \neg p \Leftrightarrow false$
- $p \vee \neg p \Leftrightarrow true$
- $p \wedge (p \vee q) \Leftrightarrow p$
- $p \vee (p \wedge q) \Leftrightarrow p$

Most of these are self evident. The last two bear some explanation. How does one convince oneself that the following is correct?

$$p \wedge (p \vee q) \Leftrightarrow p$$

One way to do this is to look at the case that q is true and that q is false. If q is true, then:

$p \wedge (p \vee true) = p \wedge true$ // Any predicate OR true is always true
$= p$ // Any predicate AND true is itself

In the case that q is false:

$p \wedge (p \vee false) = p \wedge p$ // Any predicate OR false is itself
$= p$ // Any predicate AND itself is itself

A similar argument is made to verify $\mathbf{p \vee (p \wedge q) \Leftrightarrow p}$.

An example of how a query processor will apply this to a query is given the following terribly messy query:

```
SELECT    ENAME
FROM      EMP, ASG
WHERE     EMP.ENO = ASG.ENO     AND
          ((TITLE = 'Manager'   OR
            DUR = 12)           AND
            NOT (TITLE <> 'Manager' AND
```

```
                DUR = 12))          AND
(TITLE = 'Manager'                  OR
 ENAME = 'Smith')
```

The query processor will assign the following predicates:

p => TITLE = ‘Manager’
q => DUR = 12
r => ENAME = ‘Smith’

The predicate clause is represented as:

predicate	=	$((p \vee q) \wedge \neg(\neg p \wedge q)) \wedge (p \vee r)$	// Restatement of constraint
	=	$((p \vee q) \wedge (p \vee \neg q)) \wedge (p \vee r)$	// DeMorgan’s law
	=	$(p \vee (q \wedge \neg q)) \wedge (p \vee r)$	// Associative law
	=	$(p \vee \text{false}) \wedge (p \vee r)$	// Redundancy elimination
	=	$p \wedge (p \vee r)$	// Redundancy elimination
	=	p	// Redundancy elimination

The query processor will thus simplify the query to become:

```
SELECT  ENAME
FROM    EMP, ASG
WHERE   EMP.ENO = ASG.ENO  AND
        TITLE = 'Manager'
```

The query is now in a form that is more efficient for the query processor to execute than in its initial form.

Rewriting Queries

After normalizing and simplifying queries, the query processor is now ready to start transforming the query into an efficient query plan. The technique is to choose a representation of query statements that can also be used to supply a query execution processor with a query execution plan. We will also need rules to transform query representations into new representations that maintain the fidelity of the original query statements while providing a query execution processor with a more efficient query execution plan.

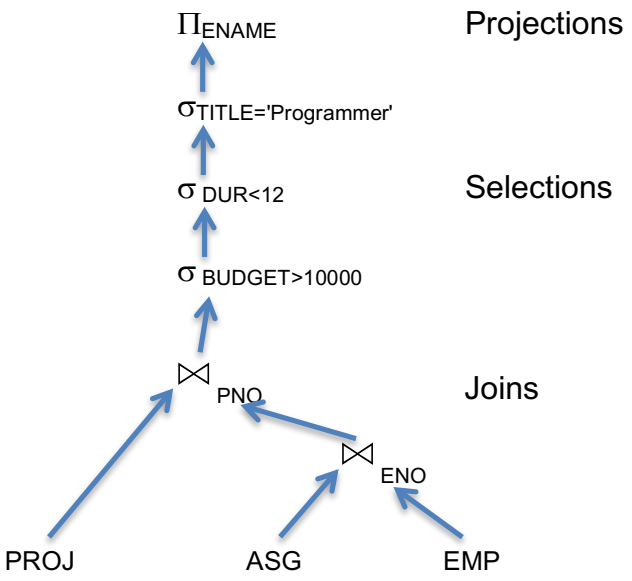
Given the following query:

```
SELECT    ENAME
FROM      EMP, ASG, PROJ
WHERE     EMP.ENO = ASG.ENO      AND
          ASG.PNO = PROJ.PNO     AND
          TITLE = 'Programmer'   AND
          DUR < 12               AND
          BUDGET > 100000
```

we transform it to a relational algebra expression:

$$\Pi_{\text{ENAME}} (\sigma_{\text{TITLE='Programmer'} \wedge \text{DUR<12} \wedge \text{BUDGET>10000} \wedge \text{EMP.ENO = ASG.ENO} \wedge \text{ASG.PNO=PROJ.PNO}} (\text{EMP} \times \text{ASG} \times \text{PROJ}))$$

Now we transform the expression into a relational tree:



Reading from the bottom up, we notice that part of the selection predicates includes natural joins between ASG and EMP as well as between ASG and PROJ. We represent these as a natural join between the ASG and EMP tables on the ENO column and then a natural join between the results of the first join and the PROJ table on the PNO column. Moving up the structure, we represent the other constraints that are applied to the join results. Finally, we represent the projection applied to the results of the constraints.

This structure can also be used as a query plan to the query execution process. The query execution process uses the tree as instructions that describe how to solve the query. It reads the tree from the bottom to the top starting from the joins until it finishes the projects. In this case, it will interpret the tree to start with joining the ASG and EMP tables on the ENO column. It will then join the results to PROJ on the PNO column. It will then apply the constraints to filter the results. Finally, it will project the ENAME column from the results the values to the calling application.

Most likely, the initial structure of a relational tree will not represent a near optimal query plan. We need some mechanism to transform the tree into another tree representing the identical query, but a more efficient query plan. To do this, we define some transformation rules for manipulating query trees.

(Watch Module 5C video – Query Transformation Rules)

Conclusion

The query processor transforms a query statement into a query plan. The transformation stops incorrect queries from being processed and streamlines poorly formed queries into queries that are processed more efficiently. The query processor then creates a tree representation of the query that represents both the query statement and a query plan. This query plan is usually not in a form that will execute efficiently. Therefore, the query processor applies query tree transformation rules to produce a more efficient query plan.

This module covered streamlining that can be performed on queries before considering join clauses. Joins are the most expensive part of query processing. For a given number of tables to be joined, there are an exponential number of possible join plans ranging in degrees of efficiency. Finding near optimal join plans is complex for centralized databases; it is even more complex for distributed databases. The next module addresses planning efficient joins across distributed databases.