Chapter 11

Distributed Query Processing

This chapter introduces the basic concepts of distributed query processing. A query involves the retrieval of data from the database. In this chapter, different phases and sub phases of distributed query processing are briefly discussed with suitable examples. An important phase in distributed query processing is distributed query optimization. Different query optimization strategies, distributed cost model, cardinalities of intermediate result relations are focused in this chapter. Finally, efficient algorithms for centralized and distributed query optimization have also been presented in this chapter.

The organization of this chapter is as follows. Section 11.1 introduces the fundamentals of query processing and objectives of distributed query processing are presented in Section 11.2. The different steps for distributed query processing such as query transformation, query fragmentation, global query optimization, and local query optimization are briefly discussed in Section 11.3. In Section 11.4, join strategies in fragmented relations have been illustrated. The algorithms for global query optimization are represented in Section 11.5.

11.1 Concepts of Query Processing

The amount of data handled by a database management system increases continuously and it is no longer unusual for a DBMS to manage data size of several hundred gigabytes to terabytes. In this context, a critical problem in a database environment is to develop an efficient query processing technique which involves the retrieval of data from the database. A significant amount of research have been dedicated to develop highly efficient algorithms for processing queries in different databases, but hence the attention is restricted on relational databases. There are many ways in which a complex query can be executed, but the main objective of query processing is to determine which one is the most cost effective. This complicated task is performed by a DBMS module, called **query processor**. In query processing, the database users generally specify what data is required rather than the procedure to follow to retrieve the required data. Thus, an important aspect of query processing is query optimization. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.

A query is expressed by using a high-level language such as SQL (Structured Query Language) in relational data model. The main function of a relational query processor is to transform a high-level query into an equivalent lower-level query (relational algebra), and the transformation must achieve both correctness and efficiency. The lower-level query actually implements the execution strategy for the given query. In a centralized

DBMS, query processing can be divided into four main steps such as **query decomposition** (consisting of scanning, parsing and validation), **query optimization**, **code generation** and **query execution**. In query decomposition step, the query parser checks the validity of the query and then translates it into an internal form usually a relational algebraic expression or something equivalent. The Query Optimizer examines all relational algebraic expressions that are equivalent to the given query and chooses the optimum one that is estimated to be the cheapest. The Code Generator generates the code for the access plan selected by the query optimizer and the query processor actually executes the query. The steps of query processing in centralized DBMS are illustrated in **figure 11.1.**

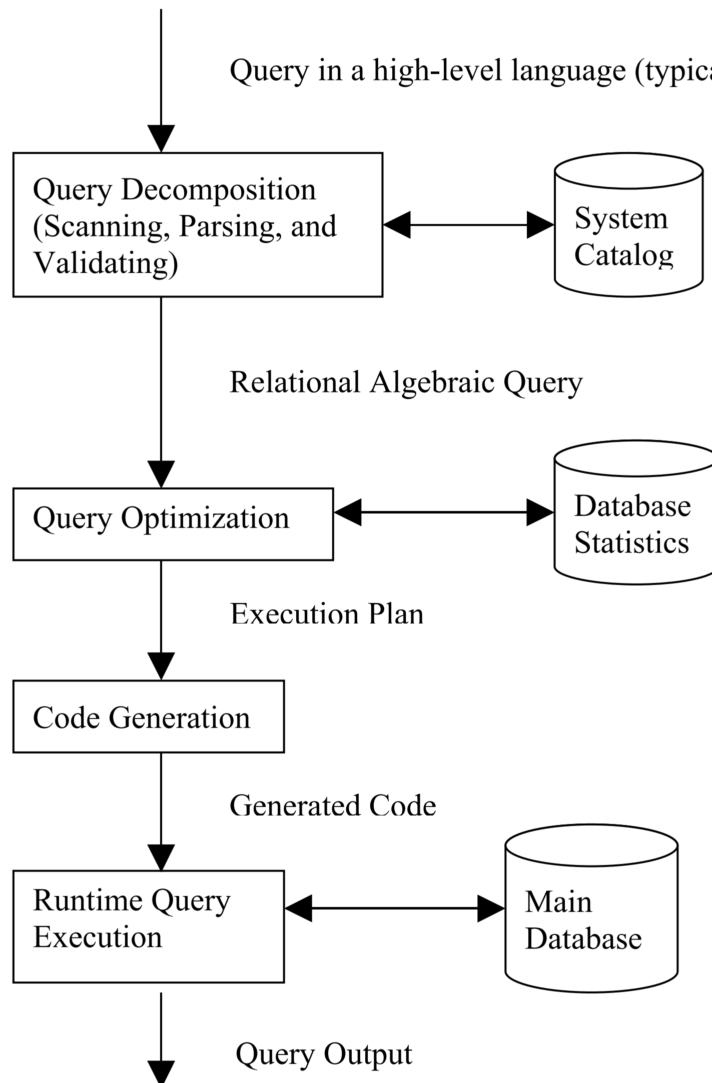


Figure 11.1 Steps in Query Processing for Centralized DBMS

The query optimization is a critical performance issue in centralized database management system, because the main difficulty is to choose one execution strategy that minimizes the consumption of computing resources. Another objective of query optimization is to reduce the total execution time of the query which is the sum of the execution times of all individual operations that make up the query. There are two main techniques for query optimization in centralized DBMS, although the two methods are usually combined in practice. The first approach uses **heuristic rules** that order the operations in a query while the second approach compares different strategies based on their relative costs and selects one which requires minimum computing resources. In a distributed system, several additional factors further complicate the process of query execution. In general, the relations are fragmented in a distributed database, therefore, the distributed query processor transform a high-level query on a logical distributed database (entire relation) into a sequence of database operations (relational algebra) on relation fragments. The data accessed by the query in a distributed database must be localized so that the database operations can be performed on local data or relation fragments. Finally, the query on fragments must be extended with communication operations, and optimization should be done with respect to a cost function that minimizes the use of computing resources such as disk I/Os, CPUs and communication networks.

Example 11.1

Let us consider the following two relations which are stored in a centralized DBMS

Employee (<u>empid</u>, ename, salary, designation, deptno)
Department (<u>deptno</u>, dname, location)

and the following query:

"Retrieve the names of all employees whose department location is 'inside' the campus"

where, **empid** and **deptno** are primary key for the relation Employee and Department respectively and **deptno** is a foreign key of the relation Employee.

Using SQL, the above query can be expressed as:

Select ename from Employee, Department where Employee.deptno = Department.deptno and location = "inside".

Two equivalent relational algebraic expressions that correspond to the above SQL statement are as follows:

(i) $\prod_{ename} (\sigma_{(location = \text{'inside'}) \wedge (Employee.deptno = Department.deptno)} (Employee \times Department))$

(ii) $\prod_{ename}(Employee \bowtie_{Employee.deptno = Department.deptno} (\sigma_{location = \text{'inside'}} (Department)))$

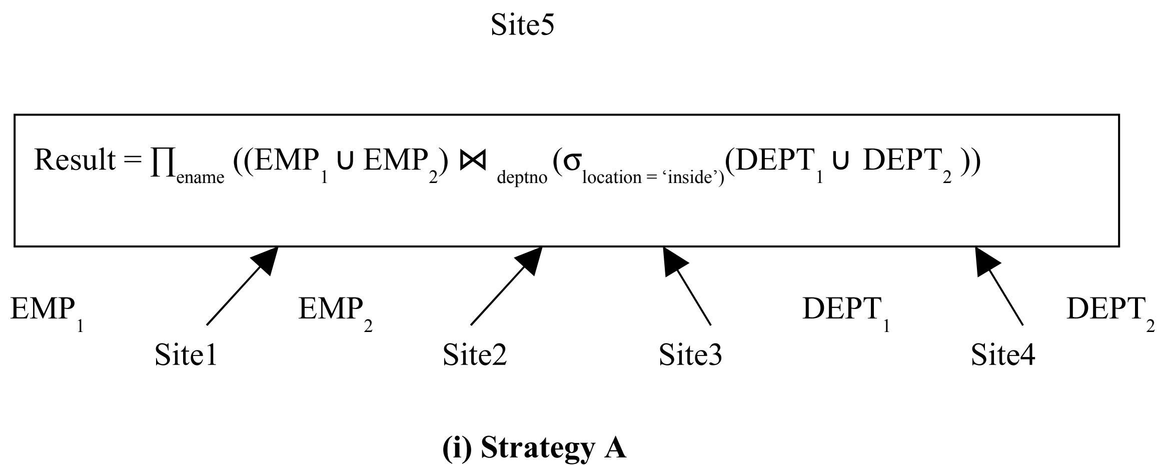
In the first relational algebraic expression the projection and the selection operation has been performed after calculating the Cartesian product of two relations Employee and Department, while in the second expression the Cartesian Product has been performed after performing the selection and the projection operation from individual relations. Obviously, the use of computing resources is lesser in the second expression. Thus, in a centralized DBMS, it is easier to choose the optimum execution strategy based on a number of relational algebraic expressions that are equivalent to the same query. In distributed context, the query processing is significantly more difficult because the choice of optimum execution strategy depends on some other factors such as data transfer among sites and the selection of best site for query execution. The problem of distributed query processing is discussed below with the help of an example.

Example 11.2

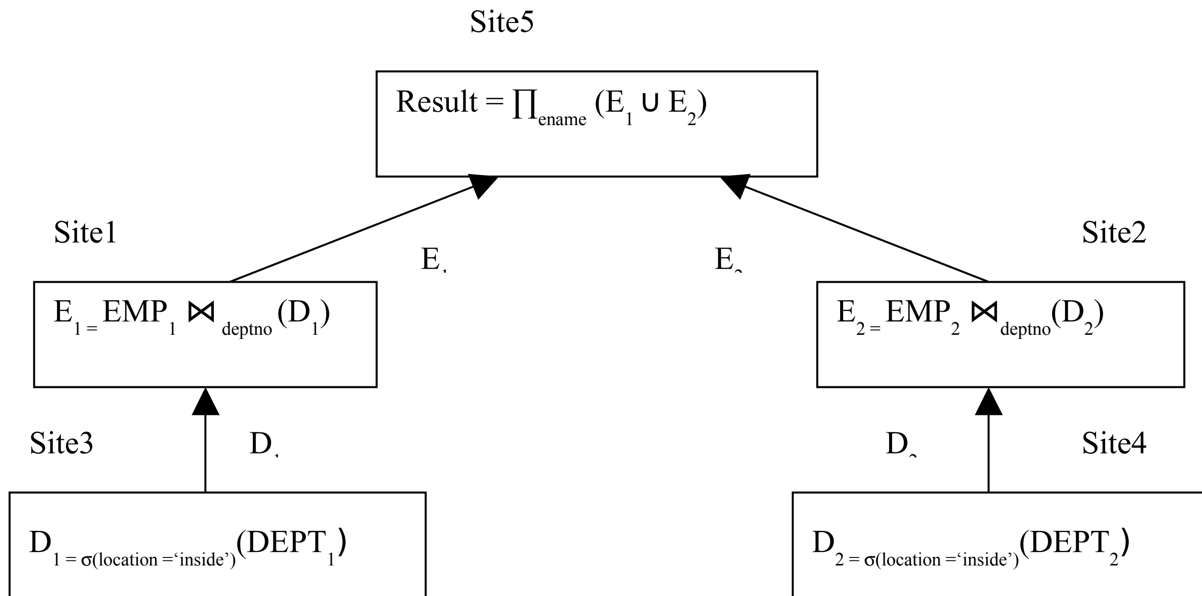
Let us consider the same example 11.1 in a distributed database environment where the Employee and Department relations are fragmented and stored into different sites. For simplicity, let us assume that the Employee relation has horizontally fragmented into two partitions EMP_1 and EMP_2 which are stored at site1 and site2 respectively and the Department relation has horizontally fragmented into two relations $DEPT_1$ and $DEPT_2$ which are stored in site3 and site4 respectively as listed below.

$$\mathbf{EMP_1 = \sigma_{deptno <= 10} (Employee)}$$
$$\mathbf{EMP_2 = \sigma_{deptno > 10} (Employee)}$$
$$\mathbf{DEPT_1 = \sigma_{deptno <= 10} (Department)}$$
$$\mathbf{DEPT_2 = \sigma_{deptno > 10} (Department)}$$

Further assume that the above query is generated at Site5 and the result is required at that site. To execute the query into the distributed environment, two different strategies are depicted in **figure 11.2.**



(i) Strategy A



(ii) Strategy B

Figure 11.2 Two Equivalent Distributed Query Execution Strategies

In the first strategy, all data are transferred into Site5 before processing the query. In the second strategy, selection operations are performed individually into fragmented relations $DEPT_1$ and $DEPT_2$ at Site3 and Site4 respectively, and then the resultant data D_1 and D_2 has been transferred at Site1 and Site2 respectively. After evaluating the join operations D_1 with EMP_1at Site1 and D_2 with EMP_2 at Site2, the resultant data has been transmitted to Site5 and the final project operation has been performed.

To calculate the costs of the above two different execution strategies, let us assume that the cost of accessing a tuple from any relation is 1 unit and the cost of transferring a tuple between any two sites is 10 units. Further consider that the number of tuples in Employee and Department relations are 1000 and 20 respectively, among which the location of 8 departments are inside the campus. For the sake of simplicity, assume that the tuples are uniformly distributed among sites and the relations Employee and Department are locally clustered on attributes 'deptno' and 'location' respectively.

The total cost for the strategy A is as follows:

(i) The cost for transferring 10 tuples of $DEPT_1$ from Site3 and 10 tuples of $DEPT_2$ from Site4 to Site5 = (10 + 10) * 10 = 200

(ii) The cost for transferring 500 tuples of EMP_1 from Site1 and 500 tuples of EMP_2 from Site2 to Site5 = (500 + 500) * 10 = 10000

(iii) The cost of producing selection operations from $DEPT_1$ and $DEPT_2$ = 20 * 1 = 20

(iv) The cost of performing join operation with EMP_1 and EMP_2 with resultant selected data from Department relation = 1000 * 8 * 1 = 8000

(v) The cost for evaluating projection operation at Site5 to retrieve the employee names = 8 * 1 = 8

In this case, the total cost is = 18,228.

The total cost for the strategy B is listed in the following:

(i) The cost of producing D_1 from $DEPT_1$ and D_2 from $DEPT_2$ = (4 + 4) * 1 = 8

(ii) The cost for transferring 4 tuples of D_1 from Site3 to Site1 and 4 tuples of D_2 from Site4 to Site2 = (4 + 4) * 10 = 80

(iii) The cost of producing E_1 at Site1 and E_2 at Site2 (4 + 4) * 1 * 2 = 16

(iv) The cost for transferring 4 tuples of E_1 from Site1 to Site5 and 4 tuples of E_2 from Site2 to Site5 = (4 + 4) * 10 = 80

(v) The cost for evaluating projection operation at Site5 to retrieve the employee names = 8 * 1 = 8

Hence, the total cost is = 192.

The cost of performing the projection operation and the selection operation on a tuple is same because in both cases it is equal to the cost of accessing a tuple. Obviously, the execution strategy B is much more beneficial than the strategy A. Furthermore, the slower communication between the sites and the higher degree of fragmentation may increases the difference among the alternative query processing execution strategies.

11.2 Objectives of Distributed Query Processing

Distributed query processing involves the retrieving of data from physically distributed databases which provides the view of a single logical database to users. It has a number of objectives as listed below.

- The major objective of distributed query processing is to translate a high-level query on a single logical distributed database (as seen by the users) into a low-level language on physically distributed local databases.

- There are numbers of alternative execution strategies for processing a distributed query. Thus, another important objective of distributed query processing is to select an efficient execution strategy for the execution of a distributed query that minimizes the consumption of computing resources.

- In distributed query processing, the **total cost** for executing a distributed query should be minimized. If no relation is fragmented in a distributed DBMS, the query execution involves only a local processing cost. On the other hand, if relations are fragmented, a communication cost is incurred in addition with the local processing cost. In this case, the aim of distributed query processing is to minimize the total

execution cost of the query which includes the total processing cost (sum of all local processing costs in participating sites) of the query and the communication cost. The local processing cost of a distributed query is evaluated in terms of the number of disk accesses (I/O cost) and CPU cost. The CPU cost is incurred when performing data operations in the main memory in participating sites. The I/O cost can be minimized by using efficient buffer management technique. In a distributed query execution, the communication cost is required to exchange data between participating sites. Hence, the communication cost depends on several factors such as the amount of data transfer between participating sites, the selection of best site for query execution, the number of message transfer between participating sites, and the communication network. In case of high-speed wide area networks (with a bandwidth few kilobytes per second), the communication cost is the dominant factor and the optimization of CPU cost and I/O cost can be ignored in such cases. The optimization of local processing cost is of greater significance in case of local networks.

- In distributed query processing, one approach to query optimization is to minimize **the total cost of time** that is required to execute a query. Another alternative approach for query optimization is to reduce **the response time** of a distributed query, which is the time elapsed between the initiation of the query and the answer of the query is produced. Therefore, another important objective of distributed query processing is to maximize the parallel execution of operations of a given distributed query and thus, the response time is to be significantly less than the total cost time.

11.3 Phases in Distributed Query Processing

In a distributed database system, data distribution details are hidden from the users, thus, it provides data distribution transparency. The distributed query processor is responsible for transforming the query on global relations into a number of subqueries on fragmented relations at local databases. Therefore, distributed query processing is more complicated than query processing on centralized DBMS, and it requires some extra steps in addition with all the steps of a centralized query processing technique. Whenever a distributed query is generated at any site of a distributed system, it follows a sequence of phases to retrieve the answer of the query. These are **query decomposition, query fragmentation, global query optimization** and **local query optimization** as illustrated in figure 11.3. The first three phases in distributed query processing are controlled by the site where the query is generated. The local query optimization is done at local participating sites and all these optimized local queries are executed on local databases. Finally, the results of local queries are sent back to the site of the distributed system where the query is generated. The details of all these phases are described in the next section.

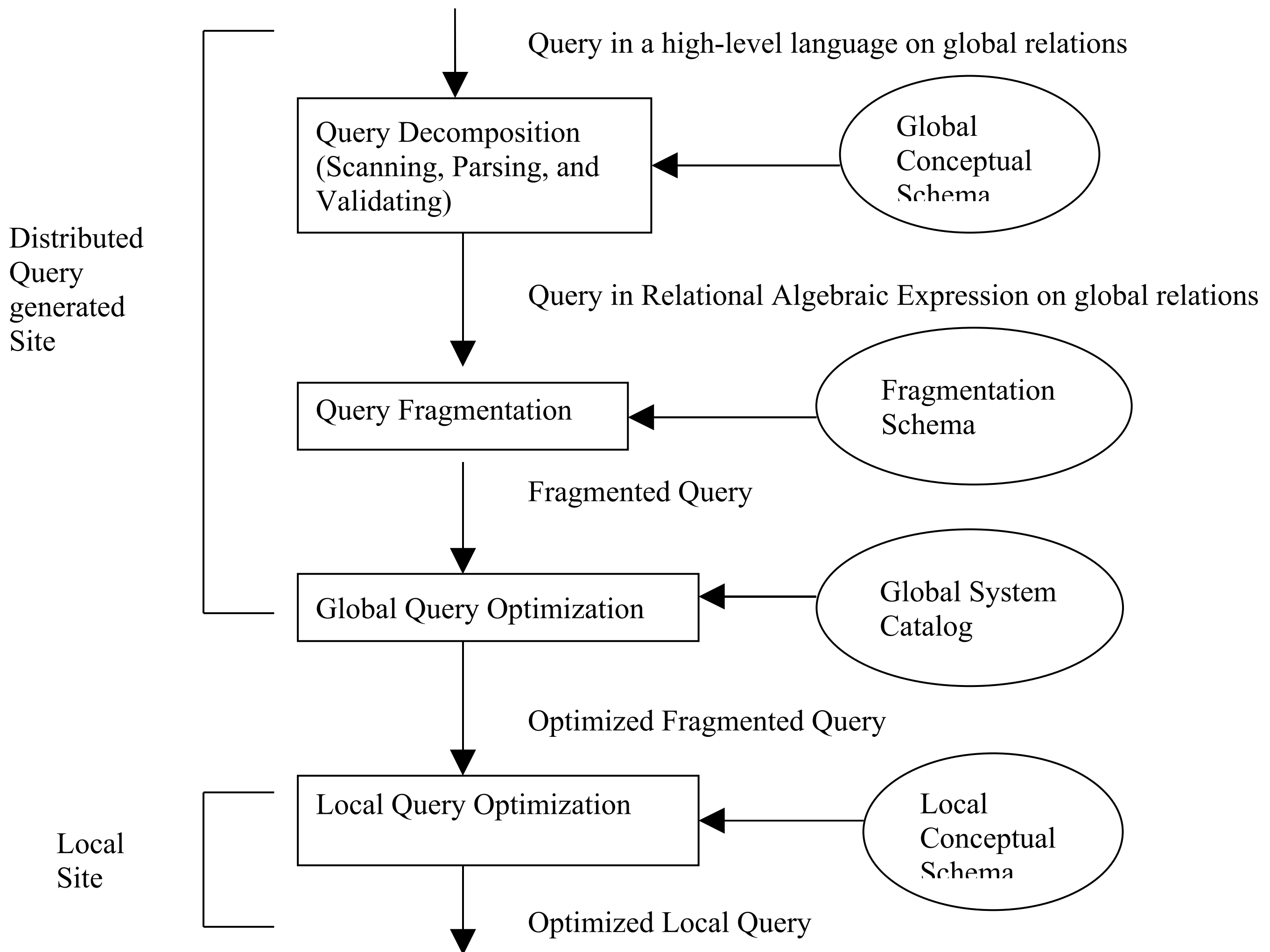


Figure 11.3 Phases in Distributed Query Processing

Query decomposition is the first phase in distributed query processing. The objective of this phase is to transform a query in high-level language on global relations into a relational algebraic query on global relations. The information required for the transformation is available in global conceptual schema describing the global relations. In this phase, the syntactical and semantic correctness of the query is also checked. In the context of both centralized DBMS and distributed DBMS, the query decomposition phase is the same. The four successive steps of query decomposition are **normalization, analysis, simplification** and **query restructuring** which are briefly discussed in the following sections.

11.3.1.1 Normalization

In normalization step, the query is converted into a normalized form to facilitate further processing in an easier way. A given query can be arbitrarily complex depending on the predicates (WHERE clause in SQL) specified in the query. In this step, the complex query is generally converted into one of two possible normal forms by using a few transformation rules, which are listed below.

- $P_1 \wedge P_2 \Leftrightarrow P_2 \wedge P_1$
- $P_1 \vee P_2 \Leftrightarrow P_2 \vee P_1$
- $P_1 \wedge (P_2 \wedge P_3) \Leftrightarrow (P_1 \wedge P_2) \wedge P_3$
- $P_1 \vee (P_2 \vee P_3) \Leftrightarrow (P_1 \vee P_2) \vee P_3$
- $P_1 \wedge (P_2 \vee P_3) \Leftrightarrow (P_1 \wedge P_2) \vee (P_1 \wedge P_3)$
- $P_1 \vee (P_2 \wedge P_3) \Leftrightarrow (P_1 \vee P_2) \wedge (P_1 \vee P_3)$
- $\neg(P_1 \wedge P_2) \Leftrightarrow \neg P_1 \vee \neg P_2$
- $\neg(P_1 \vee P_2) \Leftrightarrow \neg P_1 \wedge \neg P_2$
- $\neg(\neg P_1) \Leftrightarrow P_1$

where, P_i represents a simple predicate specified in the query.

The two possible normal forms are **conjunctive normal form** and **disjunctive normal form.**

Conjunctive Normal Form: In conjunctive normal form, preference is given to the AND ($\wedge$ predicate) operator and it is a sequence of conjunctions that are connected with the $\wedge$ (AND) operator. Each conjunction contains one or more terms connected by the $\vee$ (OR) operator. For instance,

$(P_1 \vee P_2 \vee P_3) \wedge (P_4 \vee P_5 \vee P_6) \wedge \ldots\ldots\ldots \wedge (P_{n-2} \vee P_{n-1} \vee P_n)$,

where P_i represents a simple predicate.

Disjunctive Normal Form: In disjunctive normal form, preference is given to the OR ($\vee$ predicate) operator and it is a sequence of disjunctions that are connected with the $\vee$ (OR) operator. Each disjunction contains one or more terms connected by the $\wedge$ (AND) operator. For example,

$(P_1 \wedge P_2 \wedge P_3) \vee (P_4 \wedge P_5 \wedge P_6) \vee \ldots\ldots\ldots \vee (P_{n-2} \wedge P_{n-1} \wedge P_n)$,

where P_i represents a simple predicate. In this normal form, a query can be processed as independent conjunctive subqueries connected by union operations.

Example 11.3

Let us consider the following two relations stored in a distributed database

Employee (empid, ename, salary, designation, deptno)
Department (deptno, dname, location)

and the following query:

"Retrieve the names of all employees whose designation is Manager and department name is Production or Printing".

In SQL, the above query can be represented as

Select ename from Employee, Department where designation = "Manager" and Employee.deptno = Department.deptno and dname = "Production" or dname = "Printing".

The conjunctive normal form of the query is as follows:

designation = "Manager" ∧ Employee.deptno = Department.deptno ∧ (dname = "Production" ∨ dname = "Printing")

The disjunctive normal form of the same query is

(designation = "Manager" ∧ Employee.deptno = Department.deptno ∧ dname = "Production) ∨ (designation = "Manager" ∧ Employee.deptno = Department.deptno ∧ dname = "Printing")

Hence, in the above disjunctive normal form, each disjunctive connected by ∨ (OR) operator can processed as independent conjunctive subqueries.

11.3.1.2 Analysis

The objective of the analysis step is to reject normalized queries that are incorrectly formulated or contradictory. The query is lexically and syntactically analyzed in this step by using the compiler of high-level query language in which the query is expressed. In addition, this step verifies whether the relations and attributes specified in the query are defined in the global conceptual schema or not. It is also checked in the analysis step that the operations to database objects specified in the given query are correct for the object type. When one of the above incorrectness is detected, the query is returned to the user with an explanation; otherwise, the high-level query has been transformed into an internal form for further processing. The incorrectness in the query is detected based on the corresponding **query graph** or **relation connection graph**, which can be constructed as follows:

- A node is created in the query graph for the result and for each base relation specified in the query.
- An edge between two nodes is drawn in the query graph for each join operation and for each project operation in the query. An edge between two nodes that are not result nodes represents a join operation, while an edge whose destination node is the result node represents a project operation.
- A node in the query graph which is not result node is labeled by a select operation or a self-join operation specified in the query.

The relation connection graph is used to check the semantic correctness of the subset of queries that do not contain disjunction and negation. Such a query is semantically incorrect if its relation connection graph is not connected. A **join graph** for a query is a subgraph of the relation connection graph which represents only join operations specified in the query and it can be derived from the corresponding query graph. The join graph is useful in query optimization phase. A query is contradictory if its **normalized attribute connection graph** [Rosenkrantz & Hunt, 1980] contains a cycle for which the valuation sum is negative. The construction of **normalized attribute connection graph** for a query is mentioned in the following.

- A node is created for each attribute referenced in the query and an additional node is created for a constant 0.
- A directed edge between two attribute nodes is drawn to represent each join operation in the query. Similarly, a directed edge between an attribute node and a constant 0 node is drawn for each select operation specified in the query.
- A weight is assigned to edges depending on the inequality condition mentioned in the query. A weight v is assigned to the directed edge a1→a2, if there is an inequality condition in the query that satisfies a1 ≤ a2 + v. Similarly, a weight –v is assigned to the directed edge 0→a1, if there is an inequality condition in the query that represents a1 ≥ v.

Example 11.4

Let us consider the following two relations

Student (<u>s-id</u>, sname, address, course-id, year) and
Course (<u>course-id</u>, course-name, duration, course-fee, intake-no, coordinator)

and the query "Retrieve the names, addresses and course names of all those student whose year of admission is 2008 and course duration is 4 years".

Using SQL, the above query can be represented as:

Select sname, address, course-name from Student, Course where year = 2008 and duration = 4 and Student.course-id = Course.course-id.

Hence, course-id is a foreign key of the relation Student. The above SQL query can be syntactically and semantically incorrect for several reasons. For instance, the attributes sname, address, course-name, year, duration are not declared in the corresponding schema or the relations Student, Course are not defined in the global conceptual schema. Furthermore, if the operations “=2008” and “=4” are incompatible with data types of the attributes **year** and **duration** respectively, then the above SQL query is incorrect.

The query graph and the join graph for the above SQL query is depicted in the following figures.

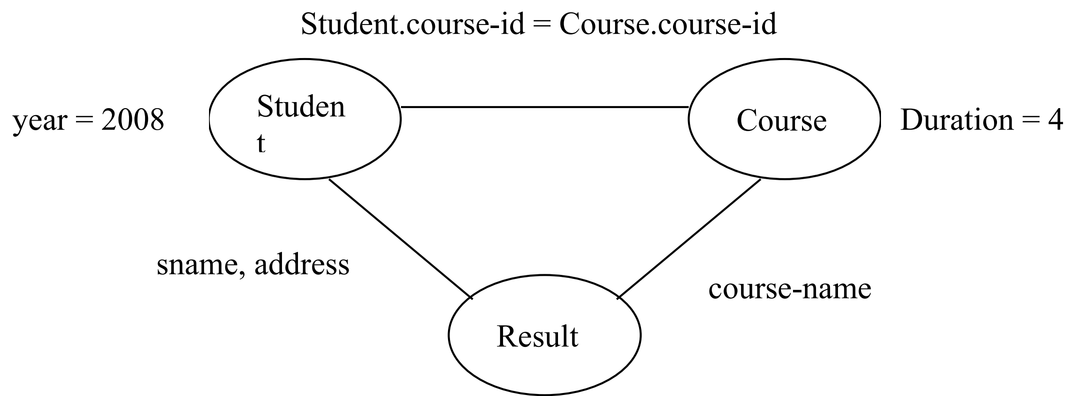


Figure 11.4(a) Query Graph

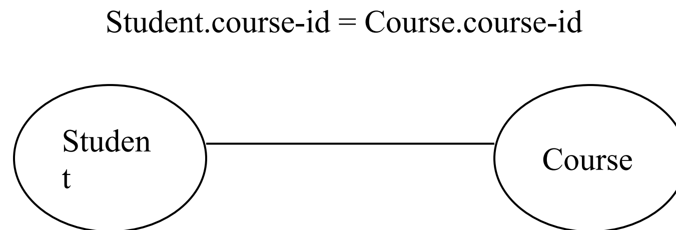


Figure 11.4(b) Join Graph

In the above SQL query, if the join condition between two relations (that is Student.course-id = Course.course-id) is missing, then there should be no line between the nodes representing the relations Student and Course in the corresponding query graph (**figure 11.4(a)**). Hence, the SQL query is semantically incorrect since the relation connection graph is disconnected. In this case, either the query is rejected or an implicit Cartesian product between the relations is assumed.

Example 11.5

Let us consider the query "Retrieve all those student names who are admitted into courses where the course duration is greater than 3 years and less than 2 years", that involves the relations Student and Course.

In SQL, the query can be expressed as:

Select sname from Student, Course where duration > 3 and Student.course-id = Course.course-id and duration < 2.

The normalized attribute connection graph for the above query is illustrated in figure 11.5.

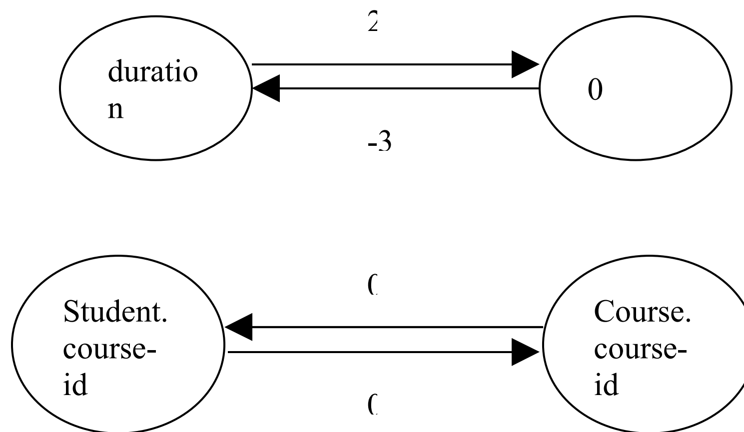


Figure 11.5 Normalized Attribute Connection Graph

In the above normalized attribute connection graph, there is a cycle between the nodes duration and 0 with a negative valuation sum, which indicates that the query is contradictory.

11.3.1.3 Simplification

In this step, all redundant predicates in the query are detected and common sub expressions are eliminated in order to transform the query into a simpler and efficient computed form. This transformation must achieve the semantic correctness. Typically, view definitions, access restrictions and integrity constraints are considered in this step, some of which may introduce redundancy in the query. The well-known idempotency rules of Boolean algebra are used in order to eliminate redundancies from the given query, which are listed below.

- $P \wedge P \Leftrightarrow P$
- $P \vee P \Leftrightarrow P$

- $P \wedge true \Leftrightarrow P$
- $P \wedge false \Leftrightarrow false$
- $P \vee true \Leftrightarrow true$
- $P \vee false \Leftrightarrow P$
- $P \wedge (\sim P) \Leftrightarrow false$
- $P \vee (\sim P) \Leftrightarrow true$
- $P \wedge (P \vee Q) \Leftrightarrow P$
- $P \vee (P \wedge Q) \Leftrightarrow P$

Example 11.6

Let us consider the following view definition and query on the view that involves the relation **Employee (empid, ename, salary, designation, deptno)**.

Create view V1 as select empid, ename, salary from Employee where deptno = 10;

Select * from V1 where deptno = 10 and salary > 10000;

During query resolution, the query will be:

Select empid, ename, salary from Employee where (deptno = 10 and salary > 10000) and deptno = 10;

Hence, the predicates are redundant and the WHERE condition reduces to "deptno = 10 and salary > 10000".

11.3.1.4 Query Restructuring

In this step, the query in high-level language is rewritten into equivalent relational algebraic form. This step involves two sub steps. Initially, the query is converted into equivalent relational algebraic form and then the relational algebraic query is restructured to improve performance. The relational algebraic query is represented by **query tree** or **operator tree** which can be constructed as follows:

- A leaf node is created for each relation specified in the query.
- A non-leaf node is created for each intermediate relation in the query that can be produced by a relational algebraic operation.
- The root of query tree represents the result of the query and the sequence of operations is directed from the leaves to the root.

In relational data model, the conversion from SQL query to relational algebraic form can be done in an easier way. The leaf nodes in the query tree are created from the FROM clause of SQL query. The root node is created as a project operation involving the result attributes from the SELECT clause specified in SQL query. The sequence of relational algebraic operations, which depends on the WHERE clause of SQL query, is directed

from leaves to the root of the query tree. After generating the equivalent relational algebraic form from the SQL query, the relational algebraic query is restructured by using **transformation rules** from relational algebra which are listed below. In listing these rules, three different relations R, S, T are considered, where the relation R is defined over the attributes A = $\{A_1, A_2, \ldots, A_n\}$ and the relation S is defined over the attributes B= $\{B_1, B_2, \ldots\ldots, B_n\}$.

(i) **Commutativity of binary operators** – Binary operators of relational algebra such as Cartesian product, join, union and intersection are commutative:

$R \times S \Leftrightarrow S \times R$ and $R \bowtie S \Leftrightarrow S \bowtie R$

Similarly, $R \cup S \Leftrightarrow S \cup R$ and $R \cap S \Leftrightarrow S \cap R$

This rule is not applicable to set difference and semijoin operations of relational algebra.

(ii) **Associativity of binary operators –** Cartesian product and natural join operation are always associative:

$(R \times S) \times T \Leftrightarrow R \times (S \times T)$ and

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

(iii) **Idempotence of Unary operators** – Several subsequent unary operations such as selection and projection on the same relation may be grouped together. On contrast, a single unary operation on several attributes can be separated into several subsequent unary operations as shown below:

$\sigma_{p(A1)}(\sigma_{q(A2)}(R) = \sigma_{p(A1) \wedge q(A2)}(R)$, where p and q denote predicates.

Similarly, $\prod_{A'}(\prod_{A''}(R)) \Leftrightarrow \prod_{A'}(R)$,

Where, A, A′ and A″ are set of attributes defined on relation R and A′ and A″ are subsets of A and A″ is a subset of A′.

(iv) **Commutativity of Selection and Projection –** Selection and projection operations on the same relation can be commuted:

$\prod_{A1,\ldots\ldots,An}(\sigma_{p(Ap)}(R)) \Leftrightarrow \prod_{A1,\ldots\ldots,An}(\sigma_{p(Ap)}(\prod_{A1,\ldots\ldots An,Ap}(R)))$,

where, Ap is not a member of $\{A_1, \ldots\ldots\ldots, An)$.

(v) **Commutativity of Selection with binary operators –** Binary operations of relational algebra such as Cartesian Product and join operations can be commuted with selection operation as follows:

$\sigma_{p(Ai)}$ (R x S) ⇔ ($\sigma_{p(Ai)}$ (R)) x S and

$\sigma_{p(Ai)}$ (R ⋈ $_{p(Aj, Bk)}$S) ⇔ ($\sigma_{p(Ai)}$ (R) ⋈ $_{p(Aj, Bk)}$S, where Ai ∈ R and Bk ∈ S.

Similarly, selection operation can be commuted with union and set difference operations if both relations are defined over the same schema:

$\sigma_{p(Ai)}$ (R ∪ S) ⇔ $\sigma_{p(Ai)}$(R) ∪ $\sigma_{p(Ai)}$ (S)

(vi) **Commutativity of Projection with binary operators** – Binary operations of relational algebra such as Cartesian Product and join operations can be commuted with projection operation as shown in the following:

$\prod_{C}$ (R x S) ⇔ $\prod_{A'}$ (R) x $\prod_{B'}$(S) and

$\prod_{C}$ (R ⋈ $_{p(Aj, Bk)}$S) ⇔ $\prod_{A'}$ (R) ⋈ $_{p(Aj, Bk)}$)$\prod_{B'}$(S),

where C= A′ ∪ B′, Ai ∈ A′, Bk ∈ B′ and A′ ⊆ A and B′ ⊆ B.

Similarly, projection operation can be commuted with union and set difference operations:

$\prod_{C}$(R ∪ S) ⇔ $\prod_{C}$(R) ∪ $\prod_{C}$(S).

Proofs of the above transformation rules are available in [Aho et. al, 1982]. By applying these rules, a large number of equivalent query trees can be reconstructed for a given query among which the most cost-effective one is chosen at the optimization phase. However, in this approach generation of excessive large number of operator trees for the same query is not realistic. Therefore, the above transformation rules are used in a methodical way in order to reconstruct the query tree for a given query. The transformation rules are used in the following sequence.

- Unary operations in the query are separated first in order to simplify the query.
- Unary operations on the same relation are grouped so that common expressions are computed only once [rule no. (iii)].
- Unary operations are commuted with binary operations [rule no. (v) & (vi)].
- Binary operations are ordered.

Example 11.7

Let us consider the following SQL query which involves the relations **Student (<u>s-id</u>, sname, address, course-id, year)** and **Course (<u>course-id</u>, course-name, duration, course-fee, intake-no, coordinator):**

Select sname, course-name from Student, Course where s-id > 0 and year = 2007 and Student.course-id = Course.course-id and duration = 4 and intake-no = 60.

The query tree for the above SQL query is depicted in the figure **11.6.**

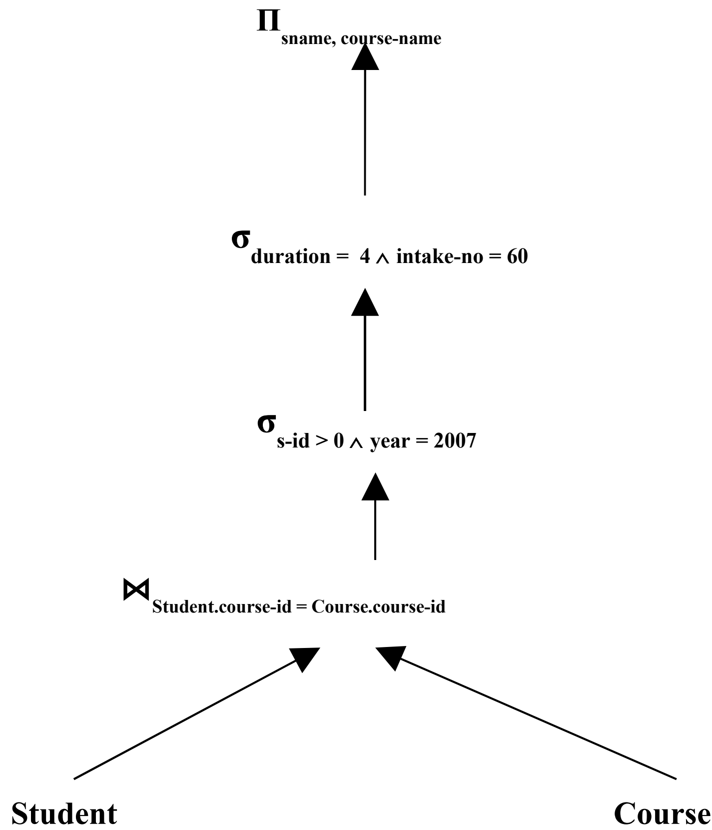


Figure 11.6 Query tree

An equivalent query tree is reconstructed by applying the above transformation rules as shown in the figure 11.7.

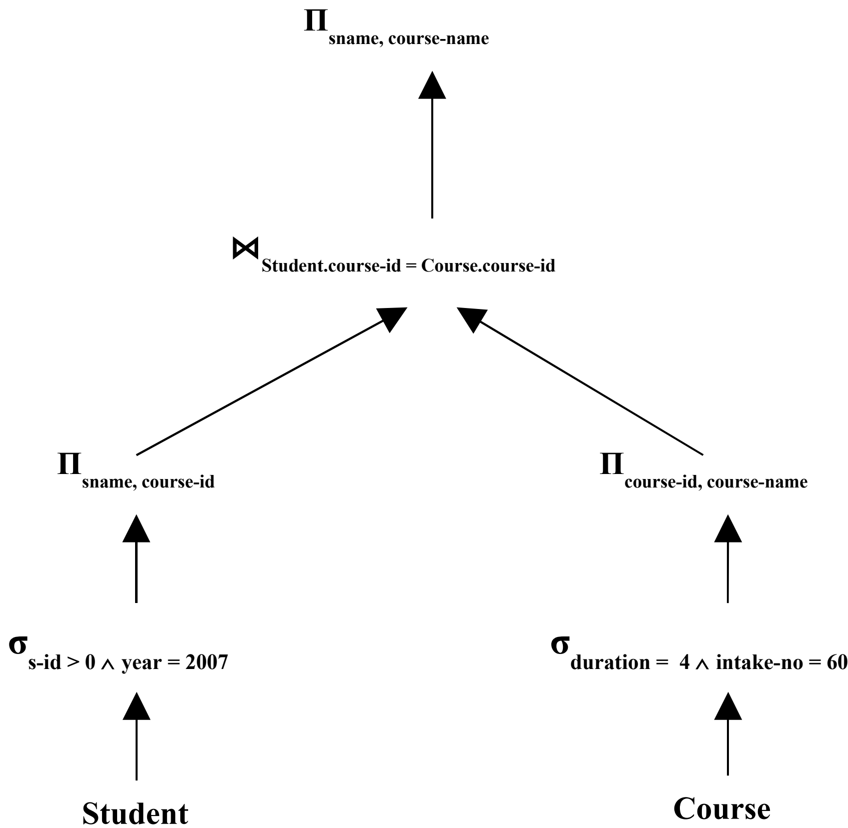


Figure 11.7 Equivalent Query tree of Figure 11.6

Hence, unary operations specified in the query are separated first and all unary operations on the same relation are group together in order to reconstruct the query tree. However, the query tree in the figure 11.7 does not necessarily imply the optimal tree.

11.3.2 Query Fragmentation

In **query fragmentation phase** a relational algebraic query on global relations is converted into an algebraic query expressed on physical fragments, called **fragment query**, considering data distribution in the distributed database. A global relation can be reconstructed from its fragments by using the **reconstruct rules** of fragmentation [as described in **Chapter 5, Section 5.3.2**]. For horizontal fragmentation, the reconstruction rule is the Union operation of relational algebra and for vertical fragmentation the reconstruction rule is the Join operation of relational algebra. Thus, query fragmentation is defined through fragmentation rules and it uses the information stored in the fragment schema of the distributed database. For the sake of simplicity, the replication issue is not considered here.

An easier way to generate fragment query is to replace the global relations at the leaves of the query tree or the operator tree of the distributed query with their reconstruction rules. The relational algebraic query tree generated by applying the reconstruction rules is known as **generic tree**. Hence, the approach is not too efficient because more simplifications and reconstructions are possible in generic trees. Therefore, in order to generate simpler and optimized query from generic query **reduction techniques** are used where the reduction techniques are dependent on the types of fragmentation. Reduction techniques for different types of fragmentation are illustrated with examples in the next section.

11.3.2.1 Reduction for Horizontal Fragmentation

In distributed database system, horizontal fragmentation is done based on selection predicates. Hence, two different reduction techniques are considered for horizontal fragmentation. These are **reduction with selection operation** and **reduction with join operation**. In the first case, if the selection predicate contradicts the definition of the fragment, then an empty intermediate result relation is produced, thus, this operation can be eliminated. In the second case, the join operation can be commuted with union operation in order to detect useless join operations in the query which can be eliminated from the result. A useless join operation exists in the query if the fragment predicates do not overlap.

Example 11.8

Let us consider the following SQL query that involves the relation **Employee (empid, ename, salary, designation, deptno)**:

Select * from Employee where salary > 10000.

Assume that the Employee relation is partition into two horizontal fragments EMP_1 and EMP_2 depending on the selection predicates as mentioned below:

$$\mathbf{EMP_1 = \sigma_{salary <= 10000} (Employee)}$$
$$\mathbf{EMP_2 = \sigma_{salary > 10000} (Employee)}$$

Now, the relation Employee can be reconstructed from its horizontal fragments EMP_1 and EMP_2 by using the following reconstruction rule.

$$\mathbf{Employee = EMP_1 \cup EMP_2}$$

Therefore, in the generic tree of the above SQL query the leaf node corresponds to the Employee relation can be replaced by the reconstruction rule $EMP_1 \cup EMP_2$. Hence, the selection predicate contradicts the definition of horizontal fragment EMP1, thereby produces empty relation. This operation can be eliminated from the generic tree as shown in **figure 11.8**.

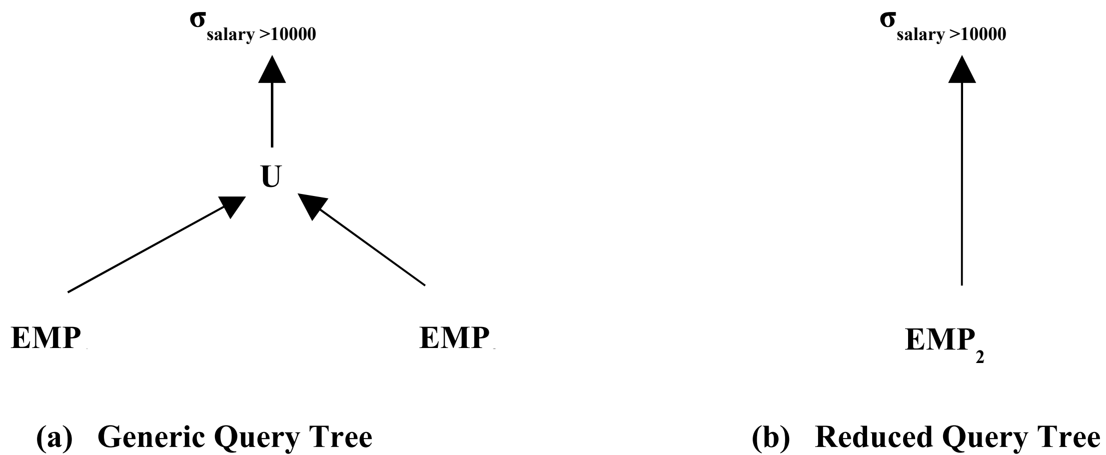


Figure 11.8 Reduction for Horizontal Fragmentation with Selection

Example 11.9

Let us assume that the relation **Employee (<u>empid</u>, ename, salary, designation, deptno)** is horizontally fragmented into two partitions EMP_1 and EMP_2 and the relation **Department (<u>deptno</u>, dname, location)** is horizontally fragmented into two relations $DEPT_1$ and $DEPT_2$ respectively. These horizontal fragmented relations are defined in the following:

$EMP_1 = \sigma_{deptno <=10} (Employee)$
$EMP_2 = \sigma_{deptno >10} (Employee)$
$DEPT_1 = \sigma_{deptno <=10} (Department)$
$DEPT_2 = \sigma_{deptno >10} (Department)$

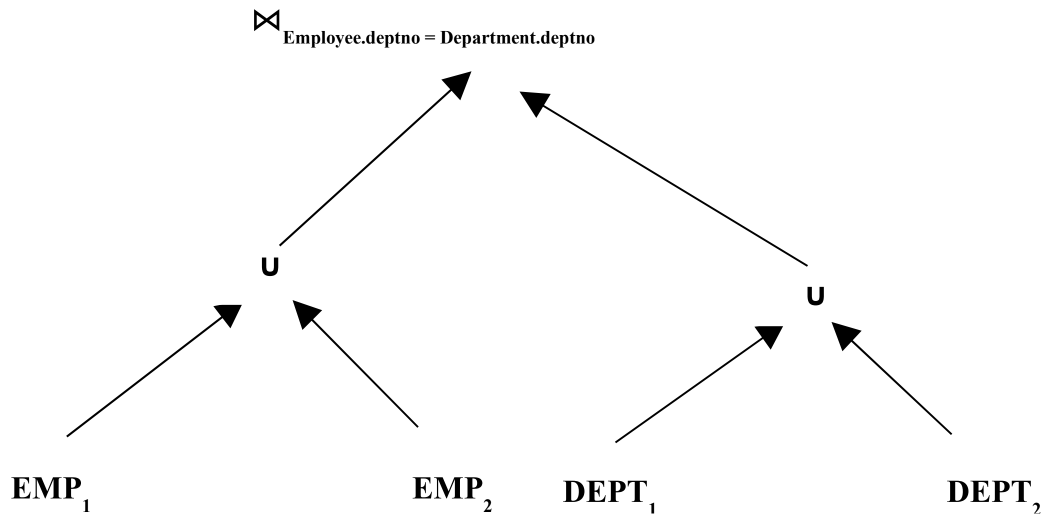
The reconstruction rules for the above horizontal fragments are as follows:

$Employee = EMP_1 \cup EMP_2$
$Department = DEPT_1 \cup DEPT_2$

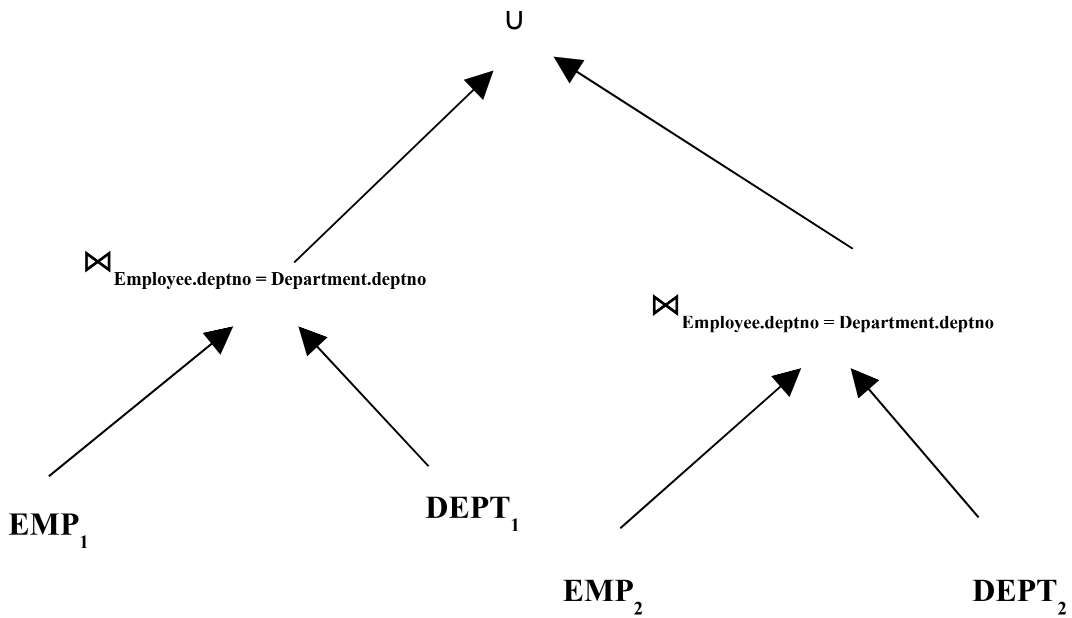
Let us consider the following SQL query in a distributed environment which involves the relations Employee and Department.

Select * from Employee, Department where Employee.deptno = Department.deptno.

The generic query tree and the reduced query tree for the above query are depicted in **figure 11.9.**



(a) Generic Query Tree



(b) Reduced Query Tree

Figure 11.9 Reduction for Horizontal Fragmentation with Join

Hence, the commutativity of join operation with union operation is very important at distributed DBMS, because it allows a join operation of two relations is to be implemented as a union operation of partial join operations, where each part of the union operation can be executed in parallel.

11.3.2.2 Reduction for Vertical Fragmentation

The vertical fragmentation partitions a relation based on projection operation, thus, the reconstruction rule for vertical fragmentation is join operation. The reduction technique for vertical fragmentation involves the removal of vertical fragments that have no attributes in common except the key attribute.

Example 11.10

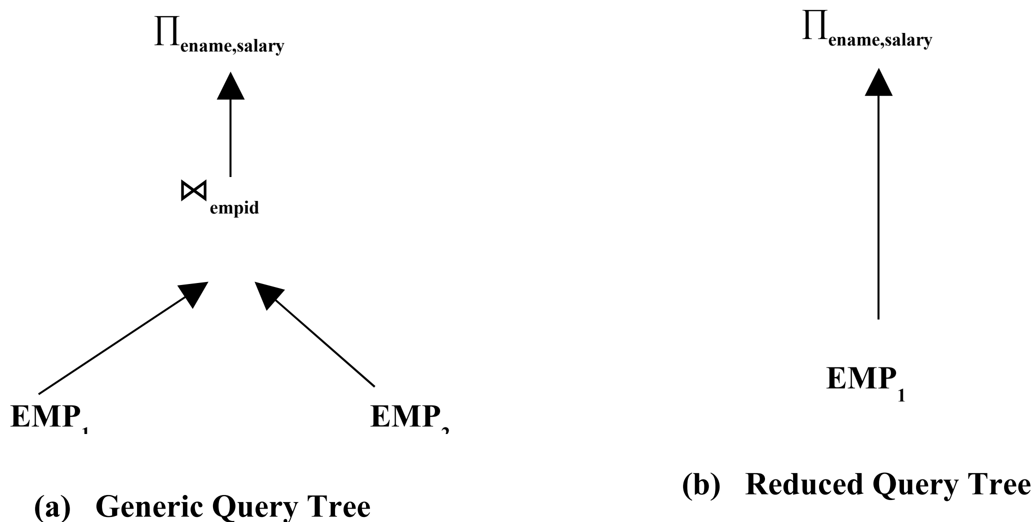
Let us assume that the Employee relation is vertically fragmented into two relations EMP_1 and EMP_2 as defined below:

$\mathbf{EMP_1 = \prod_{empid,ename,salary} (Employee)}$
$\mathbf{EMP_2 = \prod_{empid,designation,deptno} (Employee)}$

Let us consider the following SQL query:

Select ename, salary from Employee.

In this query, the projection operation on relation EMP_2 is redundant, since the attributes ename and salary are not part of EMP_2. The generic query tree of the above query is illustrated in **figure 11.10 (a).** By commutating the projection operation with join operation and removing the vertical fragment EMP_2 the reduced query tree is produced as shown in figure 11.10(b).



(a) Generic Query Tree **(b) Reduced Query Tree**

Figure 11.10 Reduction for Vertical Fragmentation

11.3.2.3 Reduction for Derived Fragmentation

The derived fragmentation partitions a child relation horizontally based on the same selection predicate of its parent relation. Derived fragmentation is used to facilitate the join between fragments. The reconstruction rule for derived fragmentation is join operation over Union operation. The reduction technique for derived fragmentation involves the commutativity of join operation with union operation and removal of useless join operations and selection operations.

Example 11.11

Let us assume that the relation **Department (<u>deptno</u>, dname, location)** is horizontally fragmented into two relations $DEPT_1$ and $DEPT_2$ respectively. These horizontal fragmented relations are defined in the following:

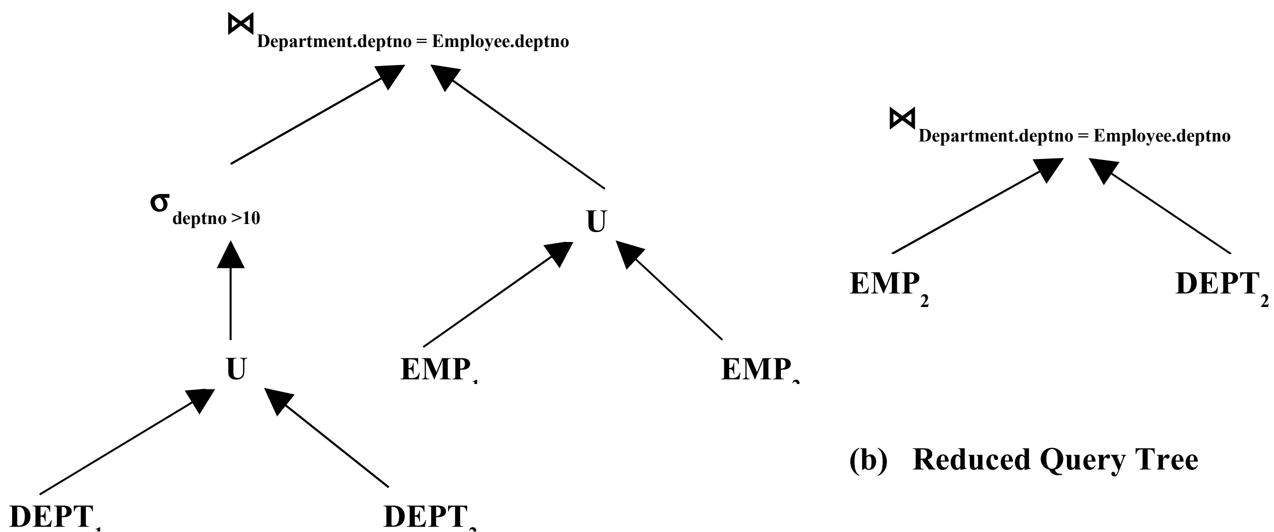
$$\mathbf{DEPT_1 = \sigma_{deptno <= 10} (Department)}$$
$$\mathbf{DEPT_2 = \sigma_{deptno > 10} (Department)}$$

Further assume that the fragmentation of Employee relation is derived from Department relation.
Thus, $EMP_i = Employee \lhd_{deptno} DEPT_i \quad i=1,2$

Let us consider the following SQL query in a distributed environment which involves the relations Employee and Department.

Select * from Employee, Department where deptno > 10 and Employee.deptno = Department.deptno.

The generic query tree for the derived fragmentation is depicted in figure 11.11(a).



(b) Reduced Query Tree

(a) Generic Query Tree

Figure 11.11 Reduction for Derived Fragmentation

In the above generic query tree, the selection operation on fragment $DEPT_1$ is redundant which can be eliminated. Similarly, since the relation $DEPT_2$ is defined depending on the selection predicate "deptno > 10", the entire selection operation can be eliminated from the above generic tree. By eliminating selection operation and commutating join operation with union operation, the reduced query tree is produced which has been shown in figure 11.11(b).

11.3.2.4 Reduction for Mixed Fragmentation

A mixed fragment is defined as a combination of horizontal, vertical and derived fragmentation. The main objective of the mixed fragmentation is to support queries that involve selection, projection and join operations. The reconstruction rule for mixed fragmentation use union operations and join operations of fragments. All reduction techniques that are used for horizontal, vertical and derived fragmentation can be used for mixed fragmentation. These reduction techniques are summarized below:

- Removal of empty relations that can be produced by contradicting selection predicates on horizontal fragments.
- Removal of useless relations that can be produced by applying projection operations on vertical fragments.
- Removal of useless join operations which can be produced by distributing join operations over union operations on derived fragments.

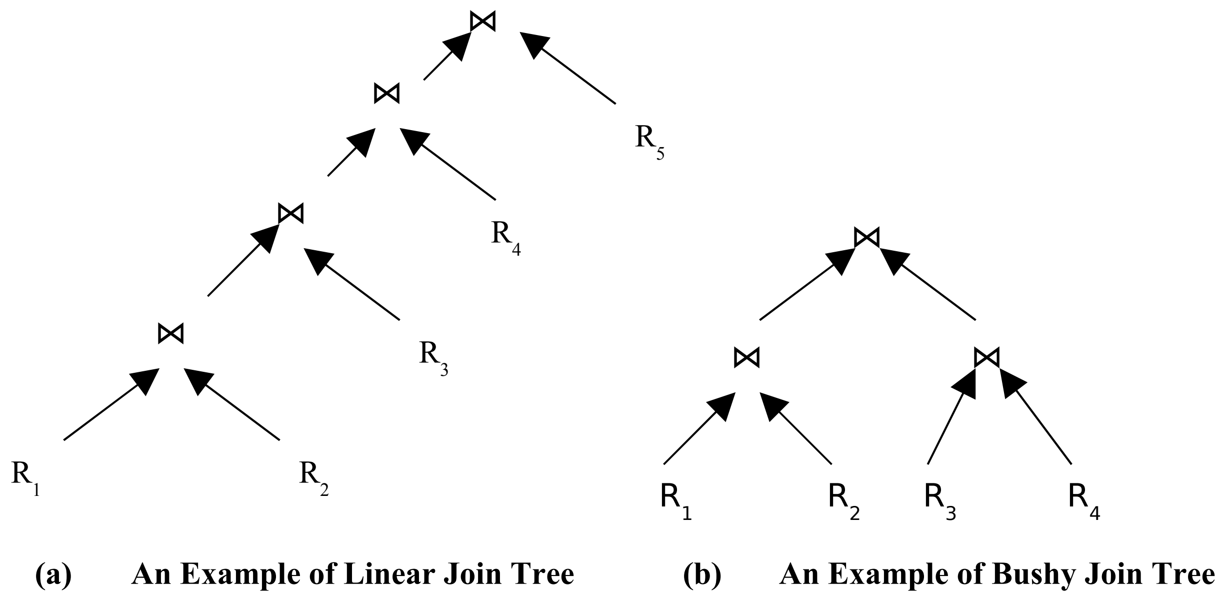
11.3.3 Global Query Optimization

Query optimization is the activity of choosing an efficient execution strategy among several alternatives for processing a query. This is same both in the context of centralized DBMS and distributed DBMS. The selected query execution strategy must minimize the cost of query processing. In a distributed DBMS, the query execution cost is expressed as a combination of local processing cost (I/O cost and CPU cost) and communication cost. To simplify the global query optimization, often the local processing cost is ignored, because the communication cost is dominant in this case. The optimal query execution strategy is selected by a software module, known as **query optimizer,** and it can be represented by three components. These are **search space, cost model** and **optimization strategy.** The search space is obtained by applying the transformation rules of relational algebra as described in Section 11.3.1.4. The cost model determines the cost of a query execution plan. The optimization strategy explores the search space and it is used to determine the optimal plan using cost model.

11.3.3.1 Search Space

The **search space** is defined as the set of equivalent query tree for a given query which can be generated by using transformation rules. In query optimization, join trees are particularly important, because it determines the join order of relations involved in a given query that effect the performance of query processing. If a given query involves many operators and many relations, then the search space is very high, because it

contains a large number of equivalent query trees for the given query. This is more expensive than the actual execution time and therefore query optimizers typically impose some restriction on the size of the search space to be considered. Most of query optimizers use heuristics rules that order the relational algebraic operations (selection, projection and Cartesian product) in the query. Another restriction is imposed on the shape of the join tree. There are two different kinds of join trees, known as **linear join trees** and **bushy join trees.** In a linear join tree, at least one operand of each operator node is a base relation. On the other hand, a bushy join tree is more general and may have operators with no base relations as operands. Linear join trees reduce the size of the search space whereas bushy join trees facilitate parallelism. The example of linear join tree and bushy join tree are illustrated in the following figure.



(a) An Example of Linear Join Tree **(b) An Example of Bushy Join Tree**

Figure 11.12 Two Different Types of Join Trees

11.3.3.2 Optimization Strategy

There are three different kinds of optimization strategies, known as static optimization strategy, dynamic optimization strategy and randomized optimization strategy, which are described in the following.

(a) **Static Optimization Strategy** - In static optimization strategy, the given query is parsed, validated, and optimized once. This strategy is similar to the approach taken by a compiler for a programming language. The advantages of static optimization strategy are the run-time overhead is removed, and there may be more time available to evaluate a large number of execution strategies, thereby increases the chances of finding a more optimum strategy. The disadvantage with the static optimization strategy is that the optimal execution strategy which is selected at compile-time may no longer be optimal at run-time.

(b) **Dynamic Optimization Strategy** – The most popular optimization strategy is dynamic optimization strategy, which is deterministic. In dynamic optimization strategy, the execution plan starts from base relations and proceeds by joining one more relation at each step until the complete plan is obtained. The advantage of dynamic optimization strategy is that it is almost comprehensive and ensures the best plan. The disadvantages are that the performance of the query is affected, because the query has to be parsed, validated and optimized each time before execution. Moreover, it is necessary to reduce the number of execution strategies to be analyzed to achieve an acceptable overhead, which may have the effect of selecting a less than optimum strategy. This strategy is suitable when the number of relations involved in the given query is small.

(c) **Randomized Optimization Strategy** - The recent optimization strategy is randomized optimization strategy, which reduces the optimization complexity but does not guarantee the best execution plan. The randomized optimization strategy allows the query optimization to trade optimization tune for execution time [Lanzelotte et al., 1993]. In this strategy, one or more start plans are built by a greedy strategy, and then it tries to improve the start plan by visiting its neighbors. A neighbor is obtained by applying a random transformation to a plan. One typical example of random transformation is exchanging two randomly chosen operand relations of the plan. The randomized optimization strategy provides better performance than other optimization strategies as soon as the query involves more than several relations.

11.3.3.3 Distributed Cost Model

The distributed cost model includes cost functions to predict the cost of operators, database statistics, base data, and formulas to calculate the sizes of intermediate results.

Cost Functions

In a distributed system, the cost of processing a query can be expressed in terms of the total cost measure or the response time measures [Yu and Chang, 1984]. The total cost measure is the sum of all cost components. If no relation is fragmented in the distributed system and the given query includes selection and projection operations, then the total cost measure involves the local processing cost only. However, when join and semijoin operations are executed, communication costs between different sites may be incurred in addition to the local processing cost. Local processing costs are usually evaluated in terms of the number of disk accesses and CPU processing time, while communication costs are expressed in terms of the total amount of data transmitted. For geographically dispersed computer networks, communication cost is normally the dominant consideration, but local processing cost is of greater significance for local networks. Thus, most early distributed DBMSs designed for wide area networks have ignored the local processing cost and concentrate on minimizing the communication cost. Therefore, the total cost measure can be represented by using the following formula.

Total cost measure = $T_{CPU} * insts + T_{I/O} * ops + C_0 + C_1 * X$

where, T_{CPU} is the CPU processing cost per instructions, insts represents the total number of CPU instructions, $T_{I/O}$ is the I/O processing cost per I/O operation, ops represents the total number of I/O operations, C_0 is the start-up cost of initiating transmission, C_1 is a proportionality constant, and X is the amount of data to be transmitted. For wide area networks, the above formula is simplified as follows.

Total cost measure = $C_0 + C_1 * X$.

The response time measure is the time from the initiation of the query to the time when the answer is produced. The response time measure must consider the parallel local processing costs and the parallel communication costs. A general formula for response time can be expressed as follows.

Response time measure = T_{CPU} * Seq_insts + $T_{I/O}$ * seq_ops + $C_0 + C_1$ * seq_X

where, seq_insts represents the maximum number of CPU instructions that can be performed sequentially, seq_ops represents the maximum number of I/O operations that can be performed sequentially, and seq_X indicates the amount of data that can be transmitted sequentially. If the local processing cost is ignored, then the above formula simplified into

Response time measure = $C_0 + C_1$ * seq_X.

In this case, any processing and communication that is done in parallel is ignored. The following example illustrates the difference between total cost measure and response time measure.

Example 11.12

Let us consider that P amount data is to be transmitted from site 1 to site 2, and Q amount of data is to be transmitted from site 3 to site 2 for the execution of a query as shown in the figure 11.13.

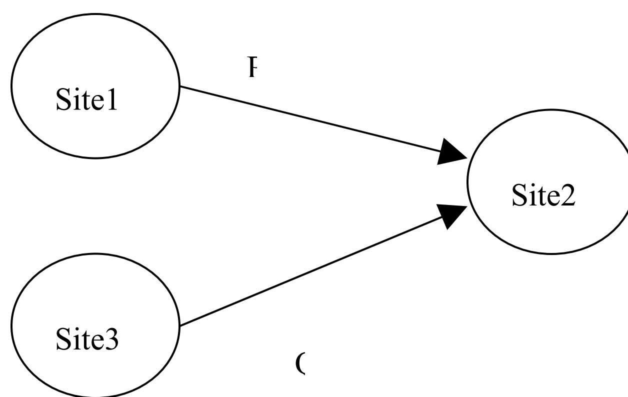


Figure 11.13 Data Transmission for a Query

Hence, the **total cost measure = $C_0 + C_1 * P + C_0 + C_1 * Q = 2C_0 + C_1 (P + Q)$.**

Similarly, the **response time measure = max $\{C_0 + C_1 * P, C_0 + C_1 * Q\}$,**

because the data transmission is done in parallel. In this case, the local processing cost is ignored. The response time measure can be minimized by increasing the degree of parallel execution, while the total cost measure can be minimized by improving the utilization of resources.

Database Statistics

The cost of an execution strategy depends on the size of the intermediate result relations that are produced during the execution of a query. In distributed query optimization, it is very important to estimate the size of the intermediate results of relational algebraic operations in order to minimize the data transmission cost, because the intermediate result relations are transmitted over the network. This estimation is done based on the database statistics of base relations stored on system catalog and formulas to predict the cardinalities of the results of the relational algebraic operations. Typically, it is expected that a distributed DBMS will hold the following information in its system catalog to predict the size of intermediate result relations.

- The length of each attribute Ai in terms of number of bytes denoted by length(Ai).
- The number of distinct values of each attribute Ai in each fragment Rj with the cardinality of the projection of fragment Rj on Ai, denoted by card($\prod_{Ai}$ (Rj)).
- The maximum and minimum possible values for the domain of each attribute Ai, denoted by Max (Ai) and Min (Ai).
- The cardinality of the domain of each attribute Ai, denoted by card(dom[Ai]), which represents the number of unique values in the dom[Ai].
- The number of tuples in each fragment Rj, denoted by card(Rj).

In addition with above information, sometimes the system catalog also holds the join selectivity factor for some pairs of relation. The join selectivity factor of two relations R and S is a real value between 0 and 1, and can be defined as follows.

SFj(R, S) = card(R $\bowtie$ S) / card(R) * card(S).

Cardinalities of Intermediate Results

To simplify the evaluation of the cardinalities of intermediate results, two assumptions have been made here. First, the distribution of attribute values in a base relation is uniform and the second is all attributes are independent, that is, the value of one attribute does not affect the values of others. The formulas to evaluate the cardinalities of the results of basic relational algebraic operations are listed in the following.

1. Selection Operation.

The cardinality of selection operation for the relation R is

$$\mathbf{card(\sigma_F (R)) = SF_S (F) * card(R)}$$

where SF_S (F) is dependent on the selection formula. The SF_S(F)s can be calculated as follows.

$\mathbf{SF_S (A = value) = 1 / card(\prod_A(R))}$

$\mathbf{SF_S (A > value) = max(A) – value / max(A) – min(A)}$

$\mathbf{SF_S (A < value) = value – min(A) / max(A) – min(A)}$

$\mathbf{SF_S (p(Ai) \wedge p(Aj)) = SF_S (p(Ai) * SF_S (p(Aj)))}$

$\mathbf{SF_S (p(Ai) \vee p(Aj)) = SF_S (p(Ai) + SF_S (p(Aj)) – (SF_S (p(Ai)) * SF_S (p(Aj)))}$

$\mathbf{SF_S (A \in \{values\}) = SF_S (A + value) * card(\{values\})}$

In this case, Ai and Aj are two different attributes, and p(Ai) and p(Aj) denotes the selection predicates.

2. Projection Operation

It is very difficult to evaluate the cardinality of an arbitrary projection operation. However, it is trivial in some cases. If the projection of relation R is done based on a single attribute, the cardinality is the number of tuples when the projection is performed. If one of the projected attributes is a key of the relation R, then

$\mathbf{card(\prod_A(R)) = card(R).}$

3. Cartesian Product Operation

The cardinality of the Cartesian product of two relations R and S is denoted as
$\mathbf{card(R \times S) = card(R) * card(S).}$

4. Join Operation

There is no general way to evaluate the cardinality of the join operation with additional information about the join operation. Typically, the upper bound of the cardinality of the join operation is the cardinality of the Cartesian product. However, there is a common case in which the evaluation is simple. If the relation R with its attribute A is joined with the relation S with its attribute B via an equi-join operation, where A is a key of the

relation R, and B is a foreign key of relation S, then the cardinality of the join operation can be evaluated as follows.
$card(R \bowtie_{A=B} S) = card(S)$.

In the above evaluation, it is assumed that the each tuple of relation R participates in the join operation; therefore, the above estimation is an upper bound. In other cases, it can be calculated as
$card(R \bowtie S) = SF_J * card(R) * card(S)$

5. Semijoin Operation

The selectivity factor of the semijoin operation of the relation R by the relation S selects the fraction of tuples of R that join with tuples of S. An approximation for the semijoin selectivity factor is represented as follows [Hevner and Yao, 1979].

$SF_{SJ} (R \ltimes_A S) = card(\prod_A (S)) / card(dom[A])$

The above formula depends on only the attribute A of the relation S. Thus, it is often called the selectivity factor of attribute A of S and is denoted by **$SF_{SJ} (S.A)$.** Now, the cardinality of semijoin operation can be expressed by the following formula.

$card(R \ltimes_A S) = SF_{SJ} (S.A) * card(R)$

This formula is applicable for common case where the attribute R.A is a foreign key of the relation S. In this case, the semijoin selectivity factor is 1, because $\prod_A(S)$ = card(dom[A]).

6. Union Operation

It is very difficult to evaluate the cardinality of union operation. The simple formulas for evaluating the upper bound and lower bound cardinalities of union operation are listed in the following.
$card(R \cup S)$ (upper bound) = $card(R) + card(S)$
$card(R \cup S)$ (lower bound) = max {card(R), card(S)}

7. Set Difference Operation

Like union operation, the formulas for evaluating the upper bound and lower bound cardinalities of set difference operation, denoted by card(R-S), are card(R) and 0 respectively.

11.3.4 Local Query Optimization

In the context of distributed query processing, local query optimization is utmost important, because the distributed query is fragmented into several subqueries and each of which is processed at local site in a centralized way. Moreover, the query optimization techniques for centralized DBMS are often extended for using in distributed DBMS. There are several query optimization techniques for centralized DBMS. One such centralized query optimization technique used by popular relational database system, INGRES [Stonebraker et al., 1976] has been introduced in the following section.

11.3.4.1 INGRES Algorithm

INGRES uses a dynamic query optimization strategy that partitions the high-level query into smaller queries recursively. In this approach, a query is first decomposed into a sequence of queries having a unique relation in common. Each of these single relation queries are then processed by a one-variable query processor (OVQP). The OVQP optimizes the access to a single relation by selecting, based on the selection predicates, the best access method to that relation. For instance, if there is a selection predicate in the query of the form <A = value>, an index available on attribute A will be used. This algorithm first executes the unary operations and tries to minimize the sizes of intermediate results before performing binary operations.

To perform decomposition, two basic techniques are used in INGRES algorithm, known as **detachment** and **substitution**. The query processor uses the detachment technique to partition a given query into several smaller queries based on a common relation. For example, using detachment technique, the SQL query q1 can be decomposed into two smaller queries q11 and q12 as follows.

q1: Select $R_2.A_2$, $R_3.A_3$, ……., $R_n.A_n$
        from R_1, R_2, ………, R_n
        where $P_1(R_1.A_{1'})$ and $P_2(R_1.A_1, R_2.A_2, \ldots\ldots R_n.A_n)$

q11: Select $P_1.A_1$ into $R_{1'}$
        from R_1
        where $P_1(R_1.A_{1'})$

q12: Select $R_2.A_2$, $R_3.A_3$, ……., $R_n.A_n$
        from $R_{1'}$, R_2, ………, R_n
        where $P_2(V_1.A_1, V_2.A_2, \ldots\ldots V_n.A_n)$

where A_i and $A_{i'}$ represent lists of attributes of relation R_i, P_1 is a selection predicate involving the attribute$A_{1'}$ from the relation R_1 and P_2 is a selection predicate involving attributes of relations R_1, R_2,……..R_n. This step is necessary to reduce the size of relations before performing binary operation. Detachment technique extracts the selection operations, which are usually the most selective ones.
Multi-relation queries that cannot be further detached are called **irreducible**. A query is said to be irreducible if and only if the corresponding query graph is a chain with two nodes. Irreducible queries are converted into single relation queries by tuple substitution.

In tuple substitution, for a given n-relation query, the tuples of one variable are substituted by their values, thereby generating a set of (n-1) variable queries. Tuple substitution can be implemented in the following way. Assume that the relation R is chosen for tuple substitution in an n-relation query q1. For each tuple in R, the attributes refers to q1 are replaced by their actual values in tuple and thereby producing a query q1` with n-1 relations. Therefore, the total number of queries produced by tuple substitution is card(R). An example of INGRES algorithm is illustrated in the following.

Example 11.13

Let us consider the following SQL query that involves three different relations **Student(<u>sreg-no</u>, sname, street, city, course-id), Course(<u>course-id,</u> cname, duration, fees),** and **Teacher(<u>T-id</u>, name, designation, salary, course-id).**

Select sname, name from Student, Course, Teacher where Student.course-id = Course.course-id and Course.couse-id = Teacher.course-id and duration = 4.

Using detachment technique, the above query can be replaced by the following queries, q1 and q2, where Course1 is an intermediate relation.

q1: **Select Course.course-id into Course1 from Course where duration = 4.**

q2: **Select sname, name from Student, Course1, Teacher where Student.course-id = Course1.course-id and Course1.course-id = Teacher.course-id.**

Similarly, the successive detachment of q2 may generate the queries q21 and q22 as follows.

q21: **Select name, Teacher.course-id into Teacher1 from Teacher, Course1 where Teacher.course-id = Course1.course-id.**

q22: **Select sname from Student, Teacher1 where Student.course-id = Teacher1.course-id.**

Assume that in query q22, three tuples are selected where course-id are C01, C03, and C06. The tuple substitution of Teacher1 relation produces three one-relation subqueries which are listed below.

q221: **Select sname from Student where Student.course-id = 'C01'.**

q222: **Select sname from Student where Student.course-id = 'C03'.**

q223: **Select sname from Student where Student.course-id = 'C06'.**

11.4 Join Strategies in Fragmented Relations

The ordering of join operation is very important both in the context of centralized and distributed query optimization. The algorithms represented in this section do not consider explicitly the fragmentation of relation. Thus, the generic term ‘relation’ is used here to denote either fragmented or nonfragmented relations. There are two basic approaches for join strategies in fragmented relations. One is **simple join strategy** which tries to optimize the ordering of join operations directly. Another alternative approach is **semijoin strategy**, which replaces join operations by semijoin operations in order to minimize communication costs.

11.4.1 Simple Join Strategy

The objective of simple join strategy is to optimize the ordering of join operations of relations in order to minimize the query processing cost. For the sake of simplicity, only specific join queries are considered here whose operand relations are stored at different sites, and it is assumed that relation transfers among different sites of the distributed system are to be done in a set-at-a-time basis rather than a tuple-at-a-time basis. Finally, the transfer time for generating the data at a result site is ignored here.

Let us consider a simple query that involves the joining of two relations R, and S which are stored at different sites. In performing $R \bowtie S$, the smaller relation is to be transfer to the site of larger relation. Therefore, it is necessary to calculate the size of the relations R and S. If the query involves one more relation T, then in that case the obvious choice is to transfer the intermediate result $R \bowtie S$ or the relation T which one is smaller.

The difficulty with the simple join strategy is that the join operation may reduce or increase the size of the intermediate results. Hence, it is necessary to estimate the size of results of join operations. One solution is to estimate the communication costs of all alternative strategies and select the best one for which the communication cost is minimize. However, the number of alternative strategies will increase as the number of relation increases.

11.4.2 Semi-Join Strategy

The main drawback of simple join strategy is that the entire operand relation is to be transmitted between the sites. The objective of the semijoin strategy is to minimize the communication cost by replacing join operations of a query with semijoin operations. The join operation $R \bowtie S$ between two relations R and S over the attribute A which are stored at different sites of the distributed system can be replaced by semijoin operations as follows.

$R \bowtie_A S \Leftrightarrow (R \ltimes_A S) \bowtie_A S \Leftrightarrow R \ltimes_A S \bowtie_A (S \ltimes_A R) \Leftrightarrow R \ltimes_A S \bowtie_A S \ltimes_A R.$

It is necessary to estimate the cost of above semijoin operations to understand the benefits of semijoin operation over join operation. The local processing costs are not considered here for simplicity. The join operation $R \bowtie_A S$ and the semijoin operation $(R \ltimes_A S) \bowtie_A S$ can be implemented in the following way assuming the relations R and S are stored at site1 and site2 respectively, and size(R) < size(S).

$R \bowtie_A S$:

2. The relation R is to be transferred at site2.
3. The join operation is performed at site2.

$(R \ltimes_A S) \bowtie_A S$:

1. The projection operation $\prod_A(S)$ is performed at site2 and result is sent to site1.
2. The site1 computes $R \ltimes_A S$, say T.
3. The result T is transferred to site2.
4. The join operation $T \bowtie_A S$ is computed at site2.

The communication cost for the above join operation is $C_0 + C_1 * size(R)$, while the communication cost for the semijoin operation is $2C_0 + C_1 * (size(\prod_A(S)) + size(R \ltimes_A S)$. Thus, the second operation is beneficial if $size(\prod_A(S)) + size(R \ltimes_A S)$ is less than size(R), because C_0 is negligible compared to the amount of data transfer.

Generally, the semijoin operation is useful to reduce the size of operand relations involved in multiple join queries. The optimal semijoin program is called **full reducer**, which reduces a relation more than others [Chiu and Ho, 1980]. The determination of full reducers is a difficult task. One solution is to evaluate the size of reduction for all possible semijoin strategies and select the best one. Full reducers cannot be found in the group of queries that have cycles in their join graph, known as **cyclic queries**. For other group of queries, called **tree queries,** full reducers exist, but the number of alternative semijoin strategies increases as the number of relations increases which complicates the above solution.

The semi-join strategy is beneficial if a few numbers of tuples participate in the join operation, while the simple join strategy is beneficial if most of tuples participate in the join operation, because semi-join strategy involves an additional data transfer cost.

Example 11.14

Let us consider the following join operation that involves three different relations **Student, Course,** and **Teacher,** and over the attribute course-id.

Student $\bowtie_{course\text{-}id}$ Course [illegible] Teacher

Further assume that the relations Student, Course and Teacher are stored at site1, site2 and site3 respectively. The c[illegible] join graph is depicted in figure 11.14.

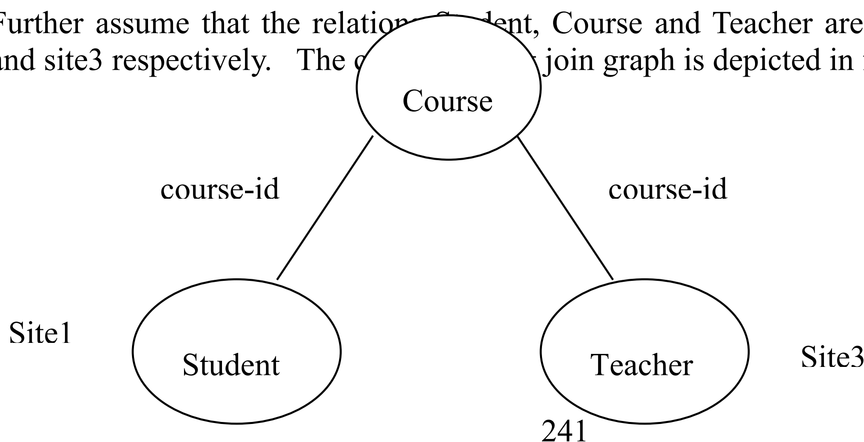


Figure 11.14 Join Graph for a Distributed Query

There are various ways to perform the above join operation, but to select the best one some information must be known. These are size(Student), size(Teacher), size(Course), size(Student ⋈ Course) and (Course ⋈ Teacher). Moreover, all data transfer can be done in parallel.

If the above join operation is replaced by semijoin operations, then the number of operations will be increased but possibly on smaller operands. In this case, instead of sending the entire Student relation to site2, only the values for the join attribute course-id of Student relation will be sent to site2. If the length of the join attribute is significantly less than the length of the entire tuple, then semijoin has good selectivity in this case, and it reduces the communication cost significantly. This is also applicable for performing join operation between the relations Course and Teacher. However, the semijoin operation may increase the local processing time, because one of the two operand relations must be accessed twice.

11.5 Global Query Optimization Algorithms

This section introduces three global query optimization algorithms, namely, distributed INGRES algorithm, R* algorithm and SDD-1 algorithm. All these algorithms are different from one another in a major way depending on several factors such as objective function, optimization strategy, network topology and so on. The comparison between these three algorithms is shown in the following.

Algorithms	Objective function	Optim. Strategy	Comm. Schemes	Optim. Factor	Semijoins	Fragments
Distributed INGRES	Total cost measure or response time measure	Dynamic	General or broadcast	Msg. transfer cost, processing cost	No	Horizontal

SDD-1	Total cost measure	Static	General	Msg. transfer cost	Yes	No
R*	Total cost measure	Static	General or local	Msg. transfer and amount of data transfer cost, I/O cost, CPU cost	no	No

Figure 11.15 Comparison of Global Query Optimization Algorithms

11.5.1 Distributed INGRES Algorithm

Distributed INGRES algorithm is an extension of centralized INGRES algorithm, thus, uses the dynamic optimization strategy. The main objective of this algorithm is to reduce total cost measure as well as response time measure, although it may be conflicting. For instance, increasing communication time by means of parallelism may decrease response time significantly. This algorithm supports horizontal fragmentation, and both general and broadcast networks can be used here. The particular site in the distributed system where the algorithm is executed is called **master site**, and it is the site where the query is initiated. The distributed INGRES algorithm works in the following way.

For a given query, all monorelation queries (unary operations such as selection and projection) can be detached first and these are processed locally. Then the reduction algorithm [Wong and Youssefi, 1976] is applied to the original query and it produces two different kinds of subqueries, irreducible subqueries and monorelation subqueries. Reduction is a technique that separates all irreducible subqueries and monorelation subqueries from the original query by detachment technique. Since monorelation subqueries are already processed locally, no action is taken for monorelation subqueries. Assume that the reduction technique produces a sequence of irreducible subqueries q1→q2→…..→qn, with at least one relation is common between two consecutive subqueries. It is mentioned in [Wong and Youssefi, 1976] that such a sequence is unique.

Based on the sequence of irreducible subqueries and the size of each fragment, one subquery is chosen from the list, say qi, which has at least two variables. For processing of the subquery qi, initially the best strategy is determined for the subquery. This strategy is expresses by a list of pairs (F, S), where F denotes a fragment which is to be transfer to the processing site S. After transferring all fragments to the corresponding processing sites, finally the subquery qi is executed. This procedure is repeated for all irreducible subqueries in the sequence q1→q2→…..→qn, and the algorithm terminates. This algorithm is represented in the following.

Distributed INGRES Algorithm:

Input: Given multi-relation query, q.
Output: Result of the last subquery qn.

Begin

Step1: Detach all monorelation queries from the given query q and execute by OVQPs (One-Variable Query Processors) at local sites same as centralized INGRES algorithm.

Step2: Apply reduction technique to q, which will produce a list of monorelation subqueries and a sequence of irreducible subqueries q1→q2→…..→qn.

Step3: Ignore all monorelation subqueries (since they are already processed by OVQPs at local sites).

Step4: for I = 1 to n, Repeat Step5 to Step8 [n is the total number of irreducible subqueries].

Step5: Choose the irreducible subquery qi involving the smallest fragments.

Step6: Determine the best strategy, pairs of (F, S), for qi. [F represents a fragment and S represents the processing sites.]

Step7: For each pair (F, S), transfer the fragment F to the corresponding processing site S.

Step8: Execute the query qi. Check for the termination of for loop.

Step9: Output the result.

End.

The query optimization is basically done in step5 and step6 in distributed INGRES algorithm. In this algorithm, subqueries are produced depending on several components and their dependency order. Since the relation involved in a subquery may be fragmented and stored at different sites, the subquery cannot be further subdivided. The main difficulty in step6 is to determine how to execute the subquery by selecting the fragments that will be transferred to sites where the processing will take place. For a n-relation subquery, fragments from n-1 relation must be transferred to the site(s) of fragments of the remaining relation, and then replicated there. Further, the remaining relations may be divided into several equal fragments in order to increase parallelism. This approach is called **fragment-and-replicate**, which performs a substitution of fragments rather than of tuples like centralized INGRES algorithm. The selection of remaining relations and the number of processing sites depends on the objective function and the communication scheme. The choice of number of processing sites is a trade-off between the total time measure and the response time measure. If the number of sites increases, the response time decreases by parallelism, but the total time increases which leads to higher communication cost. The cost of producing the result is ignored here.

The distributed INGRES algorithm is characteristics by a limited search of the solution space, an optimization decision taken for each step without concerning the consequences of that decision on global optimization. The exhaustive search approach is an alternative to the limited search approach in which all possible strategies are evaluated to determine the best one. However, dynamic optimization strategy is beneficial, because the exact sizes of the intermediate result relations are known.

11.5.2 Distributed R* Algorithm

The objective of distributed R* algorithm [Selinger and Adiba, 1980] is to reduce the total cost measure which includes the local processing costs and communication costs. Distributed R* algorithm uses an exhaustive search approach to select the best strategy. Although exhaustive search incurs an overhead, but it can be repay if the query is executed frequently. The implemented version of distributed R* algorithm does not support fragmentation or replication, thus, it involves relations as basic units. This algorithm chooses one site as **master site** where the query is initiated. The query optimizer at master site can take all intersite decisions, such as the selection of the execution sites, the fragments which will be used, and the method for transferring data. The other participating sites that stored the relations involved in the query, called **apprentice sites**, can make the remaining local decisions and generate local access plans for the query execution. Distributed R* algorithm is implemented in the following way.

Distributed R* Algorithm:

Input: Query tree for the given query
Output: Optimum execution strategy (cost is minimum)
Begin
Step1: For each base relation Ri in the query tree, Repeat Step2 to Step3.
Step2: Find each access path of Ri and determine the cost of each access path.
Step3: Determine the access path of the minimum cost.
Step4: For I = 1 to n, repeat Step5.
Step5: For each order (Ri1, Ri2, ……, Rin) of the relation Ri, build the best strategy (..((APi1, …., ⋈ Ri2) ⋈ Ri3) ⋈ ……..⋈ Rin) and compute the cost for the strategy.
Step6: Select the strategy with minimum cost.
Step7: For J = 1 to M, repeat step8. [M is the total number of sites in the distributed system storing a relation involved in the query]
Step8: Determine the local optimization strategy at site J and Send.
End.

In this algorithm, based on database statistics and formulas used to estimate the size of intermediate results, and access path information, the query optimizer determines the join ordering, the join algorithms and the access path for each fragment. In addition, it also determines the sites of join results and the method of data transfer between sites. For performing the join between two relations, there may be three candidate sites. These are the site for the first relation, the site for the second relation, and the site for the result relation. Two intersite data transfer techniques are supported by distributed R* algorithm, called **ship-whole** and **fetch-as-needed.** In ship-whole technique, the entire relation is moved to the join site and stored in a temporary relation before performing the join operation. If the join algorithm is merge join, the incoming tuples are processed in a pipeline mode as they arrived at the join site. In this case, the relation does not require to store in the temporary relation. The fetch-as-needed technique is same as semijoin operation of the internal relation with external node. In this method, the external relation is sequentially scanned and for each tuple the join value is sent to the site of the internal

relation. The internal tuples that match with the value of the external tuple is selected there and then sends the selected tuples to the site of the external relation.

Ship-whole technique requires larger data transfer but fewer messages transfer than fetch-as-needed technique. Obviously, ship-whole technique is beneficial for smaller relation. On the other hand, the fetch-as-technique is beneficial if the relation is larger and the join operation has good selectivity. In fetch-as-needed technique, there are four possible join strategies for joining of the external relation with the internal relation over an attribute value. These join strategies and their corresponding costs are calculated in the following, where LC denotes the local processing cost that involves CPU cost and I/O cost, CC represents the communication cost, and A denotes the average number of tuples of internal relation S that matches with the value of one tuple of external relation R.

Strategy 1: Ship the entire external relation to the site of the internal relation.
In this case, the tuples of external relation R can be joined with the tuples of internal relation S, as they arrive. Therefore,

Total cost measure = LC (retrieving card(R) tuples from R) + CC (size(R)) + LC (retrieving A tuples from internal relation S) * card(R).

Strategy 2: Ship the entire internal relation to the site of the external relation.
In this case, the tuples of internal relation S cannot be joined with the tuples of external relation R, as they arrive. The entire internal relation S is to be stored in a temporary relation. Hence,

Total cost measure = LC (retrieving card(S) tuples from S) + CC (size(S)) + LC (storing card(S) tuples in T) + LC (retrieving card(R) tuples from R) + LC (retrieving A tuples from T) * card(R).

Strategy 3: Fetch tuples of the internal relation as needed for each tuple of the external relation.
In this case, for each tuple in R, the join attribute value is sent the site of the internal relation S. Then the A number of tuples from S that matches with this value are retrieved and sent to the site of R to be joined as they arrive. Therefore,

Total cost measure = LC (retrieving card(R) tuples from R) + CC (length (join attribute value)) * card(R) + LC (retrieving A tuples from S) * card(R) + CC (A * length (S)) * card(R).

Strategy 4: Move both relations to a third site and compute the join there.
In this case, the internal relation is first moved to the third site and stored in a temporary relation. Then, the external relation is moved to third site and the join is performed as they arrive. Hence,

Total cost measure = LC (retrieving card(S) tuples from S) + CC (size(S)) + LC (storing card(S) tuples in T) + LC (retrieving card(R) tuples from R) + CC (size(R)) + LC (retrieving A tuples from T) * card(R).

The cost of producing final result is not considered here. In case of distributed R* algorithm, the complexity increases as the number of relation increases, since it uses the exhaustive search approach based on several factors such as join order, join algorithm, access path, data transfer mode, result site etc.

11.5.3 SDD- 1 Algorithm

SDD-1 query optimization algorithm [Bernstein et al., 1981] is derived from the first distributed query processing algorithm '**hill-climbing**'. In hill-climbing algorithm, initially a feasible query optimization strategy is determined and this strategy is refined recursively until no more cost improvements are possible. The objective of this algorithm is to minimize an arbitrary function which includes the total time measure as well as response time measure. This algorithm does not support fragmentation and replication, and does not use semijoin operation. The 'hill-climbing' algorithm can works in the following way, where the input to the algorithm is query graph, location of relations involved in the query and relation statistics.

First, this algorithm selects an initial feasible solution which is a global execution schedule that includes all intersite communication. It is obtained by computing the cost of all the execution strategies that transfer all the required relations to a single candidate result site and selecting the minimum cost strategy. Assume this initial solution is S. The query optimizer splits S into two strategies, S1 followed by S2, where S1 consists of sending one of the relations involved in the join operation to the site of the other relation. These two relations are joined locally and the resulting relation is sent to the chosen result site. If the addition of cost execution strategies S1 and S2 and the cost of local join processing cost is less than S, then S is replaced by the schedule of S1 and S2. This process is continued recursively until no more beneficial strategy is obtained. It is to be noted that if n-way join operations is involved in the given query, then S will be divided into n sub schedules instead of two.

The main disadvantage of this algorithm is that it involves higher initial cost, which may not produce better strategies at all. Moreover, the algorithm gets stuck at a local minimum cost solution and fails to achieve the global minimum cost solution.

In SDD-1 algorithm, lots of modifications of hill-climbing algorithm have been done. In SDD-1 algorithm, semijoin operation is introduced to improve join operations. The objective of SDD-1 algorithm is to minimize the total communication time, the local processing time and response time is ignored here. This algorithm uses the database statistics in terms of **database profiles**, where each database profile is associated with a relation. SDD-1 algorithm can be implemented in the following way.

In SDD-1 algorithm also, one initial solution is selected and it is refined recursively. One post optimization phase is added here to improve the total time of the selected solution. The main step of this algorithm consists of determining and ordering semijoin operations in order to minimize cost. There are four phases in SDD-1 algorithm, known as

initialization, selection of beneficial semijoin operations, result site selection, and **post-optimization.** In initialization phase, an execution strategy is selected that includes only local processing. In this phase, a set of beneficial semijoins BS are also produced for using in the next phase. The second phase selects the beneficial semijoin operations from BS recursively and modifies the database statistics and BS accordingly. This step is terminated when all semijoin operations in BS are appended to the execution strategy. The execution order of semijoin operations is determined by the order in which the semijoins are appended to the execution strategy. In third phase, the result site is decided based on the cost of data transfer to each candidate site. Finally, in post-optimization phase, the semijoin operations are removed from the execution strategy which involves only relations stored at result site. This is necessary because the result site is chosen after all semijoin operations have been ordered.

11.6 Chapter Summary

- Query processing involves the retrieval of data from the database both in the context of centralized and distributed DBMS. A Query processor is a software module that performs processing of queries. A distributed query processing involves four phases. These are query decomposition, query fragmentation, global query optimization and local query optimization.
- The objective of query decomposition phase is to transform a query in high-level language on global relations into a relational algebraic query on global relations. The four successive steps of query decomposition are normalization, analysis, simplification and query restructuring.
- In normalization step, the query is converted into a normalized form to facilitate further processing in an easier way. The two possible normal forms are conjunctive normal form and disjunctive normal form.
- The objective of the analysis step is to reject normalized queries that are incorrectly formulated or contradictory.
- In simplification step, all redundant predicates in the query are detected and common sub expressions are eliminated in order to transform the query into a simpler and efficient computed form.
- In query restructuring step, the query in high-level language is rewritten into equivalent relational algebraic form.
- In query fragmentation phase a relational algebraic query on global relations is converted into an algebraic query expressed on physical fragments, called fragment query, considering data distribution in distributed databases.
- A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as query optimization. Both centralized query optimization and distributed query optimization are very important in the context of distributed query processing.
- The optimal query execution strategy is selected by a software module, known as query optimizer, and it can be represented by three components. These are search space, cost model and optimization strategy.

- The search space is defined as the set of equivalent query tree for a given query which can be generated by using transformation rules.
- There are three different kinds of optimization strategies, known as static optimization strategy, dynamic optimization strategy and randomized optimization strategy.
- In a distributed system, the cost of processing a query can be expressed in terms of the total cost measure or the response time measures.

11.7 Review Questions

1. Discuss the objectives of distributed query processing.
2. Explain the phases of centralized query processing.
3. What is query decomposition? Is it same for centralized and distributed databases? Write down the steps of query decomposition.
4. Discuss the different normal forms with examples in the context of query processing.
5. Explain how a given query is analyzed in the analysis step with a suitable example.
6. Describe what actions are performed in query restructuring phase and why?
7. What are the objectives of query fragmentation phase? Explain with example how these are achieved in case of horizontal fragmentation.
8. Discuss the reduction of mixed fragmentation with example.
9. Differentiate between linear join tree and bushy join tree.
10. Compare and contrast between different optimization strategies.
11. Describe the centralized INGRES algorithm.
12. Comment on query optimization in distributed databases.
13. Why global query optimization is difficult in distributed DBMS?
14. In distributed DBMS context, compare and contrast between simple join strategy and semi-join strategy.
15. Describe the distributed query optimization algorithm for R*.
16. Explain the distributed query optimization algorithm for SDD-1.
17. How do you define cardinality, size and distinct values for a fragment? What is the effect on these parameters for a semi-join operation?
18. Explain the rationale of semi-join reduction for query optimization in distributed databases using an example.
19. What are the different matrices for estimating cost of a relational algebraic operation?
20. Comment on the following:
    “Semi-join can be used to reduce the cost of a join operation in a distributed environment”.
    “Join operation should be done after selection, projection and union operations for distributed database systems”.
    “Query graph identifies redundant relation in an SQL”.

Exercises

1. Multiple Choice Questions:

(i) Which of the following is not a step of centralized query execution?
   a. Query decomposition
   b. Code generation
   c. Query optimization
   d. Access plan
   e. None of these.

(ii) Which of the following statement is correct?
   a. Code generator produces the equivalent relational algebraic query from an high-level (SQL) query
   b. Query processor checks the validity of the high-level query
   c. Query optimizer generates the optimum execution strategy
   d. All of these.

(iii) Which of the following is an objective of distributed query processing?
   a. To reduce the total cost of time
   b. To reduce the response time
   c. All of these
   d. None of these.

(iv) Which of the following is not a phase of distributed query processing?
   a. Query decomposition
   b. Query fragmentation
   c. Local query optimization
   d. Access plan generation.

(v) Which of the following is not a step of query decomposition?
   a. Normalization
   b. Analysis
   c. Fragmentation
   d. Query restructuring.

(vi) Which of the following statement is incorrect?
   a. The objective of normalization to transform the query into normalized form
   b. The objective of the analysis step is to reject normalized queries that are incorrectly formulated or contradictory.
   c. In simplification step, the query is converted into simpler form
   d. None of these
   e. All of these.

(vii) The normalized attribute connection graph is used in
  a. Normalization step
  b. Simplification step
  c. Query restructuring step
  d. Analysis step.

(viii) Redundant predicates are eliminated in
  a. Normalization step
  b. Simplification step
  c. Query restructuring step
  d. Analysis step.

(ix) Reduction technique are used in
  a. Query decomposition
  b. Query fragmentation
  c. Global query optimization
  d. Local query optimization.

(x) Which of the following technique is used in reduction of horizontal fragmentation?
  a. Reduction with selection operation
  b. Reduction with join operation
  c. None of these
  d. All of these.

(xi) Which of the following is done in reduction of mixed fragmentation?
  a. Removal of empty relations
  b. Removal of useless relations
  c. Removal of useless join operations
  d. None of these
  e. All of these.

(xii) Which of the following is not included in the query execution cost of a distributed DBMS?
  a. CPU cost
  b. I/O cost
  c. Storage cost
  d. Communication cost.

(xiii) Which of the following component are used to represent a query optimizer?
  a. Cost model
  b. Search space
  c. Optimization strategy
  d. All of these.

(xiv) Which of the following statement is correct?

a. In a linear join tree, at least one operand of each node is a base relation
b. In a bushy join tree, all operands must be a base relation.
c. Bushy join trees reduces the size of the search space
d. Linear join trees facilitate parallelism.

(xv) Which of the following is not an optimization strategy?
a. Static
b. Dynamic
c. Randomized
d. All of these
e. None of these.

(xvi) Which of the following is included in distributed cost model?
a. Cost functions
b. Database statistics
c. Base data
d. All of these
e. None of these.

(xvii) The join selectivity factor between two relation is
a. Greater than 1
b. Less than 0
c. Is a real number between 0 and 1
d. None of these.

(xviii) Which of the following technique is not used by INGRES algorithm?
a. Reconstruction
b. Detachment
c. Substitution
d. None of these.

(xix) In simple join strategy,
a. The ordering of join operations are optimized
b. Join operations are replaced by semi-join operations
c. Semi-join operations are replaced by join operations
d. All of these.

(xx) The full reducer exists for
a. Cyclic queries
b. Tree queries
c. Both of tree and cyclic queries
d. None of cyclic and tree queries.

(xxi) The objective of distributed INGRES algorithm is to reduce

a. Total cost measure
b. Response time measure
c. Both total cost and response time measure
d. None of these.

(xxii) Which of the following data transfer technique is supported by distributed R* algorithm?
a. Ship-whole
b. Fetch-as-needed
c. Both ship-whole and fetch-as-needed
d. None of these.

(xxiii) The objective of R* algorithm is to reduce
a. Total cost measure
b. Response time measure
c. Both total cost and response time measure
d. None of these.

(xxiv) Which of the following is supported by distributed SDD-1 algorithm?
a. Fragmentation of relations
b. Replication of relations
c. Both fragmentation and replication of relations
d. None of these.

(xxv) Which of the following is supported by implementation version of distributed R* algorithm?
a. Fragmentation of relations
b. Replication of relations
c. Both fragmentation and replication of relations
d. None of these.

(xxvi) Semi-join is required to
a. Reduce network traffic
b. Reduce memory usage
c. Increased speed
d. None of these.

2. Consider the following schemas:

EMP = (ENO, ENAME, TITLE)
ASG = (ENO, PNO, RESP, DUR)
PROJ = (PNO, PNAME, BUDGET, LOC)

Further consider the following query:

Select ENAME, PNAME from EMP, ASG, PROJ where EMP.ENO = ASG.ENO and ASG.PNO = PROJ.PNO and (TITLE = 'ELECT.ENG' or ASG.PNO < 'P3').

Draw the generic query tree (canonical tree). Transform the generic query tree to an optimized reduced query tree.

3. Simplify the following query using the idempotency rules:

   SELECT ENO FROM ASG WHERE (NOT (TITLE = 'PROGRAMMER') AND (TITLE = 'PROGRAMMER' OR TITLE = 'ELECT.ENG') AND NOT (TITLE = 'ELECT.ENG')) OR ENAME = 'J. Das'.

4. Simplify the following query using the idempotency rules:

   SELECT ENO FROM ASG WHERE RESP = 'Analyst' AND NOT (PNO = 'P2' OR DUR = 12) AND PNO ≠ 'P2' AND DUR = 12, considering
   ASG (ENO, PNO, RESP, DUR).

5. Consider the following relations EMP (eno, ename, title) and ASG (eno, pno, resp, dur). Find the names of employees who are managers of any project in SQL-like syntax and relational algebra. Is the query optimized? If not optimize it.

6. Consider the following schemas:

   EMP (ENO, ENAME, TITLE)
   PROJ (PNO, PNAME, BUDGET)
   ASG (ENO, PNO, RESP, DUR)

   The relation PROJ is horizontally fragmented in

   PROJ1 = $\sigma_{PNO \leq 'P3'}$ (PROJ)
   PROJ2 = $\sigma_{PNO > 'P3'}$ (PROJ)

   Transform the following query into a reduced query on fragments.

   SELECT BUDGET FROM PROJ, ASG WHERE PROJ.PNO = ASG.PNO AND ASG.PNO = 'P4'.

7. Consider the following schemas:
   EMP (ECODE, ENAME, DESIGN, SALARY, PNO)
   PROJECT (PNO, PNAME, BUDGET, PSTATUS)

   The relations PROJECT and EMP are horizontally fragmented as follows:

   PROJ1 = $\sigma_{PNO \leq 'P3'}$ (PROJECT)
   PROJ2 = $\sigma_{PNO > 'P3'}$ (PROJECT)

EMP1 = $\sigma_{PNO \leq 'P3'}$ (EMP)
EMP2 = $\sigma_{PNO > 'P3'}$ (EMP)

Draw the generic query tree for the following query and convert into a reduced query on fragments.

SELECT PNAME FROM PROJECT, EMP WHERE PROJECT.PNO = EMP.PNO AND PSTATUS = 'Outside'.

8. Consider the following schemas:

EMP (ENO, ENAME, TITLE, SALARY)
ASG (ENO, PNO, RESP, DUR)

The relations EMP and ASG are horizontally fragmented as follows:

ASG1 = $\sigma_{ENO \leq 'E4'}$ (ASG)
ASG2 = $\sigma_{ENO > 'E4'}$ (ASG)
EMP1 = $\sigma_{ENO \leq 'E4'}$ ($\prod_{ENO, ENAME, SALARY}$ (EMP))
EMP2 = $\sigma_{ENO > 'E4'}$ ($\prod_{ENO, ENAME, SALARY}$ (EMP))
EMP3 = $\prod_{ENO, TITLE}$ (EMP)

Draw the generic query tree for the following query and convert into a reduced query on fragments.

SELECT * FROM ASG, EMP WHERE ASG.ENO = EMP.ENO AND TITLE = 'Developer'.

9. Consider the following schemas:

EMP (ENO, ENAME, TITLE)
PROJ (PNO, PNAME, BUDGET)
ASG (ENO, PNO, RESP, DUR)

Using INGRES algorithm, detachment and substitute for the query "Retrieve names of employees working on the ORACLE project" considering there are four different values for ENO, which are E1, E2, E3, and E4.