Distributed Query Optimization

Oscar Romero
Alberto Abelló Gamazo

PID_00184142

Index

Introduction 5

Objectives 7

1. Basics of Query Optimization 9
- 1.1. Centralized Architecture 11
  - 1.1.1. Semantic Optimization 11
  - 1.1.2. Syntactic Optimization 12
  - 1.1.3. Physical Optimization 17

2. Distributed Query Optimization 19
- 2.1. Syntactic Optimization 22
  - 2.1.1. Data Localization 22
  - 2.1.2. Reduction 23
  - 2.1.3. Generating Alternative Trees 26
- 2.2. Physical Optimization 26
  - 2.2.1. Generation of Execution Alternatives 28
  - 2.2.2. Parallelism 31
  - 2.2.3. Access Plan Evaluation 36

Summary 38

Self-evaluation 39

Answer key 40

Glossary 42

Bibliography 43

Introduction

This module of the subject *Database Architecture* will introduce you to distributed query processing. The query manager is the database module responsible for processing queries. This processing involves checking the query syntax, security permissions, expanding views, checking integrity constraints of the affected schema and also optimizing the query itself.

Out of all these tasks, the query optimizer takes on special importance because of its crucial role. This module is responsible for coming up with an affordable access plan (expressed in a procedural language) benefiting as much as possible from the resources available to answer the query. Whatever the access plan this module decides on, it will eventually be executed to retrieve data and answer the query.

Consequently, the query optimizer is responsible for *translating* the declarative SQL statement into a procedural program. Traditionally, this translation has been described in three main steps: semantic, syntactic and physical optimization. The semantic optimizer is responsible for rewriting the SQL query as an equivalent, more efficient one. The syntactic optimization establishes the optimal order of the algebraic operators involved, whereas the physical optimizer finds the best way to execute each operation, mainly by considering physical structures (and their access paths), and available algorithms for implementing that operation.

Producing an access plan, however, turns out to be a massive task, since considering all the possible translations is simply out of the question (note that, along with many other parameters, the access plan depends on physical structures, algorithms that implement the operators, the order of operators, data and system configuration).

Therefore, query optimizers apply heuristics to work under certain assumptions in order to make this task feasible. Consequently, the resulting access plans *tend* (in most cases, at least) to be close enough to optimum, but in any case, the optimizer does not even guarantee that the access plan will be affordable.

Distributed environments work along the same principles. However, the task gets even more difficult because there is a new player involved: the network. Shipping data through the network is generally not efficient, and distributed query optimizers try to minimize the amount of data shipped. Unfortunately, again, with the network also under consideration, many more new combinations and execution alternatives appear (e.g., which database node executes which part of the query). Distributed query optimizers try to benefit from distribution by exploiting parallelism as much as possible; however, given the inherent complexity of the problem, they handle this by introducing new heuristics and new assumptions to simplify the search space.

In this module you will be introduced to distributed query optimization, in essence, extending centralized query processing to deal with data distribution and parallelism. Although the semantic optimizer is the same as in a centralized system, the syntactic and physical optimizers are extended appropriately and thus, they will be our main focus of attention.

Objectives

The main objective of this module is to introduce distributed query optimization. Specifically:

1. Explain the two main extensions of distributed query processing as compared to a centralized version.
2. Explain how the data localization phase of a distributed query processing works.
3. Simulate the three main reduction rules a DDBMS will put into practice.
4. Enumerate two strategies used in the global optimization phase.
5. Explain the difference between data shipping and query shipping.
6. Justify the site selection for a join execution done by a DDBMS.
7. Choose between a semi-join or distributed join strategy.
8. Compute basic cost models for distributed query processing.
9. Enumerate the main principles behind a parallel database system.
10. Explain three kinds of parallel query processing and the main approaches to support them.
11. Discuss how cost models and query plan evaluation must be extended to support parallel query processing.

1. Basics of Query Optimization

SQL is a declarative language whereby users state their information needs. Being a declarative language, users state the data they want to obtain, but not the way it must be retrieved from the database management system (DBMS). Given an SQL statement, the objective of query processing is to retrieve the data necessary to accomplish that statement. The following steps take place:

1) Validation: This involves syntax and schema validation, but is also responsible for expanding views and checking permissions.

2) Optimization: Certain rules and heuristics are applied in order to come up with the cheapest access plan (to be executed in the next step).

3) Execution: This is the execution of the access plan that was decided on in the previous step. It mainly refers to disk access and applied data transformations (by means of algorithms, such as sorting or joining) that provide the desired data.

Given a query, there are many strategies that a DBMS can follow to process it and produce its answer. These strategies vary according to available algorithms and resources used (e.g., disk and memory). All of them are equivalent in terms of their final output but vary in their cost, that is, the amount of time that they need to run. This cost difference can be several orders of magnitude. Thus all DBMSs have a module that examines different alternatives and chooses the plan that needs the least amount of time. This component is called the query optimizer.

> "A query optimizer translates a query into a sequence of physical operators that can be directly carried out by the query execution engine. The output of the optimizer is called a query access plan. The goal of query optimization is to derive an efficient execution plan in terms of relevant performance measures, such as memory usage and query response time."
>
> Evaggelia Pitoura
>
> Encyclopedia of Database Systems

Note

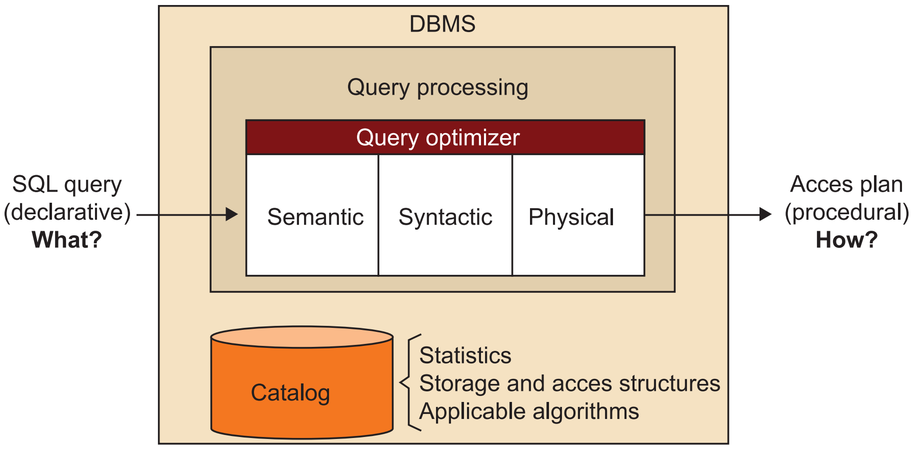
Actually, the query optimizer does not exhaustively check all execution alternatives, but uses certain heuristics to bound the exponential search space.

The optimizer, as represented in Figure 1, is a module inside the DBMS; its input is an SQL query and its output is an access plan expressed in a given procedural language. Its objective is to obtain the best possible execution algorithm (e.g., to be able to decide if an available index would perform better than a table scan). To do so, the optimizer relies on the following catalog information:

- Content statistics, such as number of tuples, size of attributes, etc.

- Available storage structures: For example, partitions and materialized views.
- Available access structures: For example, B-tree and hash indexes.
- Applicable algorithms: Mainly for joining and sorting.

Figure 1. Components of a centralized query optimizer



In general, a DBMS does not find the optimal access plan, but it obtains an approximation (in a reasonable time). Finding the optimum is computationally hard (NP-complex in the number of relations involved), potentially resulting in higher costs than just retrieving the data. Therefore, DBMSs use heuristics to reach an approximation, meaning that sometimes they do not obtain the optimum, but they are usually close to it.

It is important to know how the optimizer works in order to detect deviations and correct them, when possible (for example, adding or removing certain indexes, partitions, etc.). This is the basis of what is known as physical tuning, the main task of any DBA, which consists of deploying the best scenario (e.g., creating indexes, views, clustering, updating statistics, etc.) that produces the best access plan.

The cost function of the query optimizer typically refers to machine resources such as disk space, disk input/output, buffer space, CPU time, and network bandwidth. In current centralized systems where the database resides in disk storage, the emphasis is on minimizing the number of disk accesses.

1.1. Centralized Architecture

It is crucial to fully understand query optimization in centralized environments before tackling the same problem in distributed systems, since the same principles (with some extensions) apply for distributed query optimization. Thus, in this section, we are going to review the basics of query optimization. For now, we will ignore the presence of a network, but data may be in different disks in the same machine, as long as we do not use any kind of parallelism. Query optimization[1] is typically introduced as if executed in three sequential steps (i.e., semantic, syntactic and physical optimization).

(1)Although query optimization is discussed in three sequential modules (semantic, syntactic and physical optimization) for teaching purposes, the boundaries of these modules are not that clear in the implementation, which tends not to be that modular.

1.1.1. Semantic Optimization

This is the first step we should take and consists of transforming (i.e., rewriting) the SQL sentence into an equivalent one with a lower cost, by considering:

- Integrity constraints: Defined in the schema (e.g., checks, uniqueness, etc.)
- Logic properties: Some basic logic features can be used at this step to further exploit the schema integrity constraints (e.g., transitivity, subsumption, etc.)

In this phase, the DBMS applies transformations to a given query and produces an equivalent SQL query intended to be more efficient. For example, the predicate can be standardized in Conjunctive Normal Form (CNF) (with well-known reasoning services applied), nested queries may be flattened out in some cases, and the like. The main goal is finding tautological queries, queries with an incorrect form, or contradictory queries (always from the point of view of integrity constraints). Having an incorrect form means that there is a better way to write them from the point of view of performance, while contradictory means that their result is going to be the empty set. It is important to note that transformations performed in this phase only depend on the declarative (i.e., static) characteristics of the queries and do not take into account the actual query costs for the specific DBMS and database concerned. In other words, the input of this module is a SQL query and a schema (with a set of integrity constraints), and produces an equivalent, optimized SQL query over the same schema.

Conjunctive Normal Form

This is a predicate of the form: (x OR y) AND (z OR t OR u) AND ...

Applying transitivity and replicating clauses over equalities is a typical example of a transformation performed at this phase of the optimization to deal with incorrect queries. This may look a bit naive, but makes it possible to use indexes over both attributes. For instance, if the optimizer finds "a=b AND a=5", it will transform it into "a=b AND a=5 AND b=5", so that if there is an index over "b", it can also be taken into consideration in subsequent phases.

Another example of semantic optimization for incorrect queries is removing disjunctions. For example, we may transform the disjunction of two equalities over the same attribute into an "IN" (e.g., "a=1 OR a=2" can also be expressed as "a IN (1,2)"). Reducing the number of clauses in the selection predicate this way might also reduce its evaluation cost. However, such a transformation is not easily detected in complex predicates and can only be performed in the most simple cases.

Now, consider the following relation:

CREATE TABLE people (id INT PRIMARY KEY, age INT CHECK (age>18 AND age<65), weight FLOAT);

Any query (partially or completely) violating an integrity constraint (either from the model or from the schema) should be spotted as a contradictory query. For example:

Q1 = SELECT * FROM people WHERE id IS NULL,

Q2 = SELECT * FROM people WHERE age = 66

Q3 = SELECT * FROM people WHERE weight>50 AND weight<45

All these queries would retrieve the empty set because of different reasons. For example, Q1 is asking for NULL values on an attribute defined as the primary key (i.e., violating a model constraint that says primary keys cannot be NULL); Q2 violates a schema integrity constraint (defined with a CHECK), whereas Q3 violates the logic rules regarding numbers: a float cannot be greater than 50 and less than 45 at the same time.

In general, semantic optimization is really poor in most (if not all) DBMSs, and only useful in simple queries. Therefore, even though SQL is considered a declarative language, the way we write the statement can hide some optimizations and affect its performance. Thus, a simple way to optimize a query (without any change in the physical schema of the database) may be just rewriting it in a different way.

1.1.2. Syntactic Optimization

This consists of translating the query from SQL into a sequence of algebraic operations. There might be many sequences of algebraic operations corresponding to the same SQL query. If so, the optimizer would choose the one with minimal cost, by means of heuristics. These sequences are usually represented in relational algebra as formulas or in tree form (commonly known as a syntactic tree). The interpretation of the tree is as follows:

- Nodes, of which we distinguish three kinds:
  - Root: Represents the final output of the query and is thus unique.
  - Internal: Representing algebraic operations (e.g., joins, projections, unions, etc.).
  - Leaves: Representing table (or relation) accesses.

- Edges: Representing data flows. The node that the arrow points to requires the data produced by the node located at the tail of the arrow.

The goal of this phase is to reduce, as much as possible, the amount of data flowing from the leaves to the root by determining the order between operators. Leaves read data from tables and internal nodes transform the data read. Thus, this can be achieved mainly in two ways: (i) reducing the number of attributes as soon as possible, and (ii) reducing the number of tuples as soon as possible. From the relational algebra point of view, (i) implies projecting as soon as possible and (ii) means selecting as soon as possible.

Projecting and selecting as soon as possible is clearly desirable if we want to reduce the amount of data flowing through the algebraic operations as much as possible. Note, however, that less data does not always imply a more efficient access plan. Indexes, clusters and other access structures are only accessible at the table level (i.e., leaf level), and internal nodes (i.e., internal operators) do not benefit from them. However, access structures and join/sorting algorithms are not considered at this stage but during physical optimization.

Pushing selections and projections down the syntactic tree is a heuristic that normally leads to the optimum, but not always.

The optimizer starts creating all the alternative syntactic trees by applying these two rules:

1) Commutative property of joins: i.e., changing the order of the inputs of a given join.

2) Associative property of joins: i.e., when more than two relations are joined, commute join branches two by two.

As you already know, joins are the most expensive relational operator and thus, join order can have a big impact on performance. For example, consider the following two joins: $A \bowtie B \bowtie C$, and the following scenario:

- 90% of the tuples in B match at least one tuple from *A*,
- *B* and *A* join through a FK-PK relationship (from *A* to *B*) and statistically, we know that this relationship relates each tuple in *B* to 10 tuples in *A*, on the average,
- Only one tuple from *B* matches any tuple from *C* (and we know it exactly matches one),
- A has millions of tuples, *B* has thousands of them and *C* only contains some hundreds.

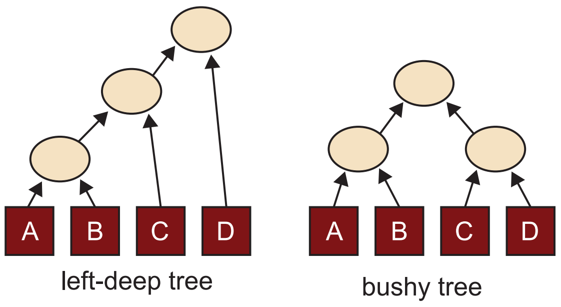
Clearly, with this information, we should decide to switch the order of the joins; that is, join C and B, and then join the result to A: $C \bowtie B \bowtie A$. For this reason, special attention must be given to how joins must be ordered. Unfortunately, the number of possible trees generated by these two rules grows, in the worst case, exponentially with the query size (i.e., number of joins). Nevertheless, most queries seen in practice involve less than 10 joins, and these two rules have proved to be worth checking in such contexts. However, for complex join queries, the number of combinations generated might be enormous.

Pipelining

To benefit from pipelining, the implementation policy of the join operator (or in general, any operator that wants to benefit from it) can be demand driven (i.e., the operator consuming data pulls from the producer) or producer driven (i.e., the producer pushes as much processed data as possible to the consumer). Typically, in order to provide pipelining, each operator must support Open, Next and Close methods. By means of these, the parent (i.e., consumer) node activates execution in the child (i.e., producer) node.

Joins to be performed in a query can always be represented as trees. In a centralized system, only left-deep (or right-deep) trees (see left-hand side of Figure 2) are generated to favor pipelining. Pipelining is a technique that uses the result of one operation as the input of the next one, without materializing it onto disk. Thus, the system does not wait for the first operation to finish in order to start the second operation. On the contrary, as soon as one tuple is obtained in the output of the first operation, it is treated by the second operation. The alternative is “materialized evaluation”, where intermediate results are stored in temporary tables.

Figure 2. Two kinds of trees



It is important to note that left/right-deep trees (also known as linear trees) are trees such that one of the join operators is always a physical table (i.e., not an intermediate result obtained from a previous operator), and therefore, centralized optimizers must only decide about the order of joins. For example, consider the first tree in Figure 2. The 4 relations, A, B, C and D, join each other (right now, how they are joined is not relevant). In this example, two relations are first joined (in this case, A and B). Next, the result obtained from this join is joined to C, which in turn is joined to D.

Bushy trees (right-hand side of Figure 2, for example) might have two intermediate inputs, producing by definition a wider search space; for this reason they are overlooked by centralized DBMSs. Note, however, that the number of linear trees we can generate is still exponential (specifically, $\Theta(2^N)$, where N is the number of tables to join).

Typically, the search space to find such trees is built by means of dynamic programming, which builds all possible plans, breadth-first, in a deterministic way. Remember that a join tree J_1 is considered to be more efficient than a join tree J_2 if the total cost of J_1 is smaller than that of J_2. Normally, the optimizer chooses between trees built according to the size of intermediate results, and this heuristic usually results in a solution not far from the optimum. Again, updated statistics about the selectivity factor of joins, table cardinalities and attribute statistics are needed to properly estimate such cost (which is not always guaranteed). In this approach, only certain heuristics are introduced to prune partial trees that are likely not to lead to the optimum solution. For this reason, this approach happens to be adequate when a few joins are involved, but it is not that appealing when many joins must be performed. In our example, this order (join *A* and *B*, then *C*, then *D*) means that, according to the optimizer's knowledge (and therefore, according to the available statistics) the result of joining *A* and *B* is the optimum in terms of size, and so on regarding the remaining partial results.

Note

When a large number of joins are involved in the query, acyclic hypergraphs are used instead of linear or bushy trees. The objective is to represent them as a finite set of semi-joins with no dangling tuples. However, large join trees fall outside the scope of this material.

In practice, there is no reason to choose between a right- or left-deep tree. Nevertheless, left-deep trees have been traditionally used to represent join trees, as a simple matter of visualization. In a left-deep tree the intermediate result produced by previous joins is always on the left side, the side that is supposed to be already in memory (i.e., in the outer loop of the nested loop join algorithm).

Next, for each alternative tree generated – it is important to consider each of the alternatives separately – the optimizer uses the following actions (based on well known equivalence rules between relational expressions) on the syntactic tree to push down selections and projections:

Note

The equivalence rules presented here are expressed as algebraic formulas, where $\sigma_A(R)$, stands for a selection over attribute *A* in relation *R*; $\pi_{A,B}(R)$, stands for the projection of attributes *A* and *B* over *R*; $R \bowtie_{A=B} S$, stands for a join between *R* and *S* on attributes *A* and *B*, respectively; and $R \cup S$, $R - S$ stand for union and difference, respectively. Alternatively, you can do the exercise to transform these formulas into syntactic trees, which are an alternative for representing queries at this stage.

- (A1) Split a selection if it uses a conjunctive predicate: $\sigma_{A=x \wedge B=y}(R) \equiv \sigma_{A=x}(\sigma_{B=y}(R))$
  - (A1.1) Alternatively, fuse two consecutive selections into one, having the conjunction of both as predicate.
- (A2) Commute the precedence of a selection and a join: $\sigma_{A=x}(R \bowtie_{C=D} S) \equiv \sigma_{A=x}(R) \bowtie_{C=D} S$, where *A* is an attribute of *R*.
- (A3) Commute the precedence of a selection and a set operation (i.e. union, intersection and difference). For example, for the union: $\sigma_{A=x}(R \cup S) \equiv \sigma_{A=x}(R) \cup \sigma_{A=x}(S)$.
- (A4) Commute the precedence of a selection and a projection, when the selection attribute is projected: $\sigma_{A=x}(\pi_A(R)) \equiv \pi_A(\sigma_{A=x}(R))$.

- (A5) Replicate the projection after a selection, adding the selection attribute, when this attribute is not projected: $\pi_B(\sigma_{A=x}(R)) \equiv \pi_B(\sigma_{A=x}(\pi_{A,B}(R)))$.

- (A6) Commute the precedence of a projection and a join, when the join attributes are projected: $\pi_{A,B}(R \bowtie_{A=B} S) \equiv (\pi_A(R)) \bowtie_{A=B} (\pi_B(S))$.

- (A7) Replicate the projection after a selection, adding the necessary join attributes, when these attributes are not projected: $\pi_{A,B}(R \bowtie_{C=D} S) \equiv \pi_{A,B}((\pi_{A,C}(R)) \bowtie_{C=D} (\pi_{B,D}(S)))$.

- (A8) Commute the precedence of projection and union: $\pi_A(R \cup S) \equiv \pi_A(R) \cup \pi_A(S)$. Note that intersection and difference do not commute. For example, given $R(A,B) = \{\{a, 1\}\}$ and $S(A,B) = \{\{a, 2\}\}$, then $\pi_A(R) - \pi_A(S) = \varnothing$, while $\pi_A(R - S) = \{\{a\}\}$.

To be precise, the optimizer applies the equivalence rules by performing the following steps:

1) Split the selection predicates into simple clauses (usually, the predicate is first transformed into CNF) by using A1.

2) Push selections towards the leaves (through joins) as much as possible, by using A2 to A4.

3) Group consecutive selections (simplify the resulting predicate if possible) by using A1.1.

4) Push projections towards the leaves (through joins and selections) as much as possible by using A5 to A8.

5) Group consecutive projections (simplify them if possible). Note that consecutive projections only appear in the presence of views or subqueries. To simplify them we just substitute both with a single projection consisting of the intersection of all attributes projected.

It is also part of syntactic optimization to detect disconnected parts of the query, if any, and simplify tautologies ($R \cap \varnothing = \varnothing$, $R - R = \varnothing$, $\varnothing - R = \varnothing$, $R \cap R = R$, $R \cup R = R$, $R \cup \varnothing = R$, $R - \varnothing = R$). Detection of disconnected tables (those with no join condition in the predicate) in a query can easily be done. However, in this case errors are not usually triggered. Instead, a cross product is performed by most (if not all) DBMSs.

Moreover, in some cases, the tree is transformed into a Directed Acyclic Graph (DAG) by fusing nodes if they correspond to exactly the same relational operation with the same parameters. For example, the same selection operation in two different subqueries of the same SQL sentence would be evaluated only once. This would be reflected by two edges going out of this selection (implying that the graph is no longer a tree).

The output of this phase is a set of syntactic trees optimized according to the equivalence rules previously presented), one for each alternative tree identified by commuting and associating join branches.

1.1.3. Physical Optimization

This is the main phase of query processing (i.e., the one that takes the most time). It takes the optimized syntactic trees produced in previous step as input. For each of these trees, it employs a search strategy that explores the space for access plans. Specifically, it needs to consider:

- Physical structures: Indexes, clusters, etc.
- Access paths: How to access data by using the available physical structures: i.e., retrieve one tuple, several tuples or full scan.
- Algorithms: Algebraic operators are no longer considered. Instead, we consider physical operations which implement the algebraic ones, plus some additional needed operations (e.g., duplicate removal, sorting or grouping).

The physical optimizer transforms each syntactic tree input into a process tree, which models the execution strategy for that syntactic tree. The process tree must be interpreted in the same way as the syntactic one, except for internal nodes that do not correspond now to an algebraic operator but rather to intermediate temporary tables generated by a physical operation (some of them with no correspondence to the relational algebra, such as grouping or sorting). The difference between the two trees is that now the nodes represent real steps to be executed. For example, most projections disappear by fusing them with the previous operation in the data flow or by just removing them if they lay on top of a physical table. Similarly, selections that do not sit on top of a physical table are normally fused with the previous physical operation.

For each process tree, the optimizer estimates the cost to execute it. To do so, it is essential to work with updated data statistics and accurate knowledge about the cost of accessing each physical structure through different paths and that of algorithms that implement each operator. Eventually, each process tree is associated with an overall execution cost; the optimizer compares all these plans and selects the overall cheapest one to be used when executing the original query.

Note

Nowadays, most optimizers apply a cost-based strategy to choose among different alternative trees produced by the syntactic optimizer. As previously explained, strong emphasis is placed on minimizing the number of disk accesses, and minimal cost is often considered to be a synonym for minimum number of disk accesses.

There are four steps in the physical optimizer:

1) Generation of alternatives: During step 1, this phase determines the implementation choices that exist for the execution of each operator specified by the syntactic tree. These choices are related to the available join algorithms (e.g., nested loops, merge join, and hash join), if/when duplicates are eliminated, and other implementation characteristics of this kind, predetermined by the DBMS implementation. They are also related to the available indexes for accessing each relation, which are determined by the physical schema of the DB.

2) Cardinality and size estimation of intermediate results: During step 2, this component estimates the size of (sub)query results and the frequency distribution of attribute values of these results, which are needed by the cost model. Most commercial DBMSs, however, base their estimation on assuming a uniform distribution (i.e., all attribute values having the same frequency).

3) Cost estimation for each algorithm and access path: During step 3, this component specifies the arithmetic formulas used to estimate the cost of access plans. For every different join method, different index-type access, and in general for every different kind of step that can be found in an access plan, there is a formula that gives an (often approximate) cost for it.

4) and, finally, choice of the best option and generation of the access plan.

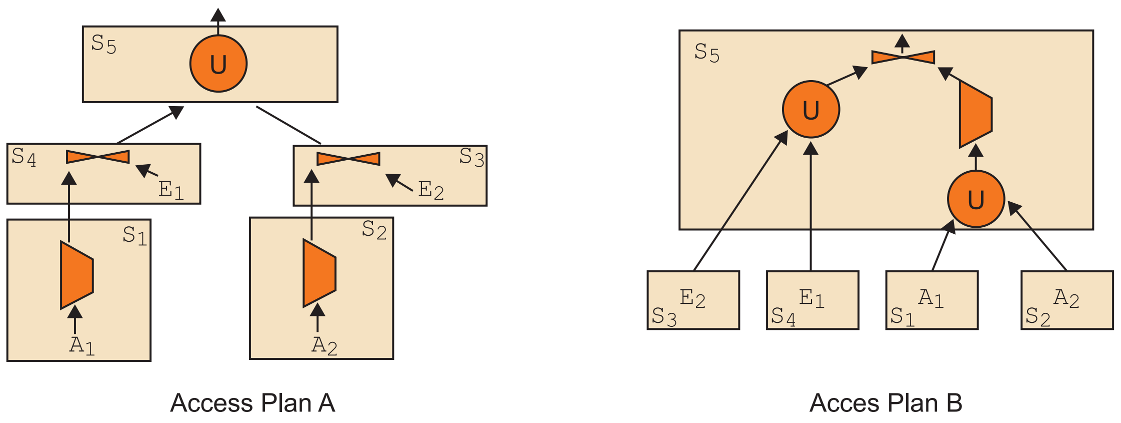
You should not underestimate the memory requirements and running time for evaluation of all the alternatives generated. Despite all the work that has been done on query optimization, there are many questions for which we do not have complete answers, even for the simplest, single-query, relational optimizations. Moreover, several advanced query optimization issues are active topics of research. These include distributed, parallel, semantic, object-oriented, and aggregate query optimization, as well as optimization with materialized views, and optimization with expensive selection predicates.

2. Distributed Query Optimization

The main goal of distributed database management systems (DDBMSs) is to hide implementation (i.e., physical) details about distribution from the users. Not only must data independence at the logical and physical level be guaranteed (inherited from the well known ANSI-SPARC architecture for centralized DBMSs), but data access must also be independent now, regardless of where data is stored (i.e., distribution transparency), or whether it is replicated or not. Consequently, the user must not be aware of the existing replicas and table partitions. Furthermore, regardless of distribution, integrity constraints must be enforced. For example, each data object must have a unique name (i.e., we can still define primary keys).

Figure 3. Example of distributed query optimization

```
SELECT *
FROM employee e, assignedTo a
WHARE e.#emp=a.#emp AND
a.responsability= 'manager';
```



```
AssignedTo (#emp,#proj, responsibility, fullTime)
         - S1: A1 = AssignedTo(#emp≤'E3')
         - S2: A2 = AssignedTo(#emp>'E3')

Employee (#emp, empName, degree)
 - S3: E2 = Employee(#emp>'E3')
 - S4: E1 = Employee(#emp≤'E3')
```

This chapter introduces how a DDBMS deals with query optimization. Distributed query optimization works along the same principles that centralized query optimization is based on, but they are extended to consider both data distribution (and the communication overhead that entails) and replication. For example, consider Figure 3, where two alternative plans for the same query are presented. There, the main issue to address (even more important than those inherited from centralized optimization) is where to execute each operation. In this example, we have five database nodes or sites (from S_1 to S_5), and four fragments (A_1, A_2, E_1 and E_2, properly defined at the bottom of the figure). Furthermore, note that each fragment is assigned to a site (in this example we are not considering replication). The two access plans differ in the order of operations (selections, joins then union versus unions, selection and then join), and in where to execute each piece of the query. In access plan A, each site is responsible for executing some query pieces, whereas access plan B mainly focuses on shipping data to site 5, which is responsible for executing all the operations. There are reasons why a distributed optimizer must pay attention to where to execute what. To comprehend the importance of this, consider the following claims and consequences related to this topic:

- In the general case, communication costs are the main parameter to minimize (perhaps not as critical for LANs), since they dominate any other parameter (including disk accesses). For example, in access plan A, the output produced in S_1 must be shipped to S_4, where it will be joined with E_1. If the size of S_1's output is orders of magnitude bigger (for example, Gigabytes) than the size of E_1 (for example, Megabytes), then the optimizer should discard this plan, because the cost of sending Gigabytes over a WAN may be unaffordable. Instead, it becomes essential to consider the alternative plan of commuting this join branch (which minimizes the communication costs, in that Megabytes instead of Gigabytes are sent over the network).

- In addition to the already complex task of choosing the join order in a centralized environment, we have to consider specific techniques to deal with this issue in distributed environments. For example, the semi-join strategy (to be introduced in section 2.2.1.) has proved to save communication costs and nicely suits distributed environments.

- Ideally, we can also exploit the distributed environment to benefit from parallelism as much as possible. For example, consider again access plan A. The selections in S_1 and S_2 can be executed in parallel and, potentially, their results could be pipelined to S_3 and S_4 where both joins could be executed in parallel.

- The distributed optimizer, in order to be able to deal with fragmentation and replication, requires a global catalog with metadata and statistics about fragments and replicas. Thus, the optimizer must know which fragments and which replicas exist and where they are located. For example, if a replica of E_1 is available in S_1, it seems reasonable to execute that join in S_1 and ignore the replica in S_4, which would imply higher communication costs.

Centralized systems put emphasis on minimizing the number of disk accesses. However, distributed systems, where replication and parallelism are cheap, put emphasis on minimizing network bandwidth usage.

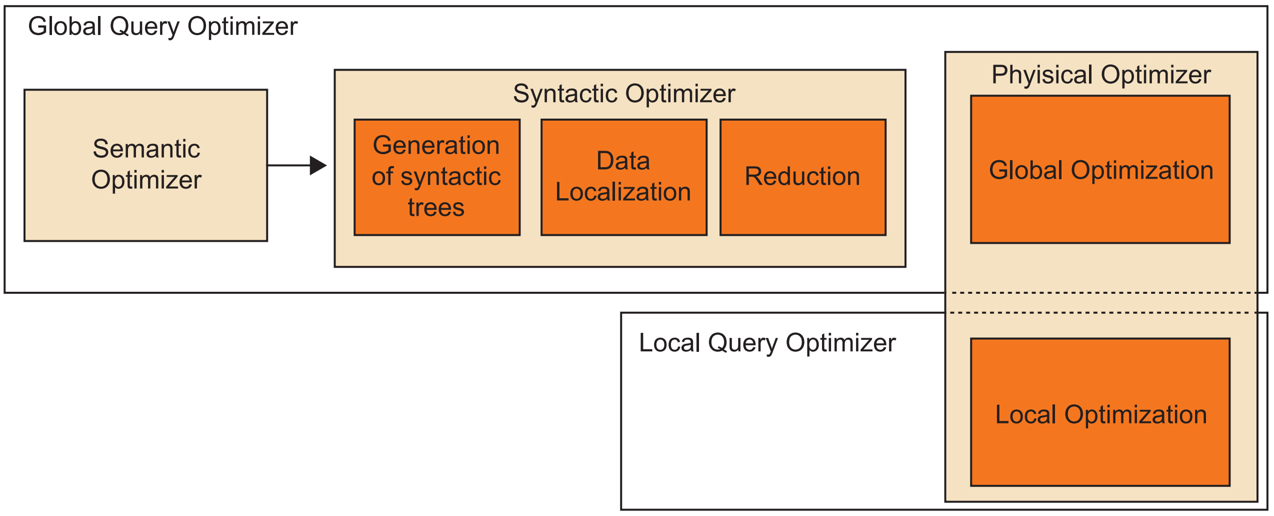
All in all, a distributed optimizer must go through some extra stages. Consider Figure 4, which outlines query optimization for distributed environments:

- Semantic optimizer: This plays the same role as in a centralized architecture (see section 1.1.1.). Thus, it tries to optimize the query simply by rewriting it into an equivalent, but more efficient, SQL query. Thus, we need not give further consideration to this component.

- Syntactic optimizer: This phase starts by representing the SQL input query in an algebraic form –typically, as a syntactic tree (see section 1.1.2.). Once the alternative syntactic trees are generated and optimized, this phase proceeds to the data location and reduction stages. The main objective of this stage is to rewrite the global query (issued in terms of the global schema) as a fragment query (data location) and identify tautological fragments that can be removed (reduction). A fragment query is posed over fragments and thus is aware of fragmentation. The fragmentation schema must be available in the global catalog in order for these stages to be executed (for further details, see section 2.1.).

- Finally, physical optimization is split into two stages:
  - Global optimization: Performed at the control site where the query was submitted, global optimization decides at which site the operation is to be executed and inserts communication primitives into the plan. It is also responsible for exploiting parallelism regarding data distribution. For this purpose, the allocation schema must be available in the global catalog (for further details, see section 2.2.).
  - Local optimization: Identical to that of centralized databases (i.e., deciding on access methods, which indexes to use, etc.). Thus, we need not give further consideration to this component.

Figure 4. Steps in distributed query optimization



2.1. Syntactic Optimization

As you know, syntactic optimization starts by describing the input query in algebraic form, normally, in the form of a syntactic tree. In a DDBMS, syntactic optimization is extended first with the data localization phase. Prior to applying equivalence rules (pushing projections and selections down as much as possible), relations referenced in the query are replaced by their corresponding fragmentation and reconstruction expressions. The result is what is known as fragment query. Once equivalence rules are applied, the process is also extended with the reduction phase, where different equivalence rules are exploited, depending on the fragmentation strategy(ies) followed in our DDBMS, in order to identify empty subtrees (i.e., that generate no results). In other words, the reduction phase identifies subtrees that are not worth being computed.

Finally, the commutative and associative join rules are applied to generate alternative trees. Realize that, in a centralized optimizer, this step was performed before applying the equivalence rules. However, in distributed systems it makes sense to swap these, so we can guarantee that the tree has been reduced as much as possible beforehand.

2.1.1. Data Localization

Note

To fully understand this section, it is highly advisable to be familiar with fragmentation techniques

Before going into detail, we need to briefly refresh some basics on fragmentation. There are two main fragmentation strategies: horizontal and vertical fragmentation. On the one hand, horizontal fragmentation can be described in terms of fragmentation predicates (i.e., selection predicates). When a relation is horizontally fragmented in terms of fragmentation predicates defined over another relation (to which it must be related), this is known as derived horizontal fragmentation. On the other hand, vertical fragmentation can be described in terms of projections. All this information, the fragments produced and where they are placed is kept in the fragmentation schema (stored in the global catalog). Furthermore, note that any fragmentation is correct if it is complete, disjoint and if the original relation can be reconstructed from the fragments produced. Horizontally fragmented relations are reconstructed by means of unions and vertically fragmented by means of joins.

Starting from the fragment schema (accessible through the global catalog), the syntactic optimizer transforms a given query into a fragment query by means of query expansion. Specifically, a global relation R is replaced by an expression that reconstructs it from its fragments, and which depends on the kind of fragmentation. In the case of horizontal fragmentation, fragments of R are united to produce the global relation, and in the case of vertical fragmentation, they are properly joined to produce R.

For example, consider the following global query:

We want to retrieve the data corresponding to the following algebraic expression:

$$Q_1 : = \sigma_{A>150}(R)$$

Let us suppose we have a horizontal fragmentation of *R*, which can be described with the following fragmentation predicates over attribute A.

$$R_1 = \sigma_{A<100}(R)$$

$$R_2 = \sigma_{100 \leq A \leq 200}(R)$$

$$R_3 = \sigma_{A>200}(R)$$

Query expansion would replace *R* by the union of these three fragments, resulting in:

$$Q_1' : = \sigma_{A>150}(R_1 \cup R_2 \cup R_3)$$

Next, the syntactic optimizer applies equivalence rules to optimize the resulting expressions. This phase has already been discussed under traditional syntactic optimization, introduced in section 1.1.2. However, in distributed environments, some extensions must be added to the traditional approach. First, a new equivalence rule must be considered in order to push joins down through unions. Consequently, a sixth step is added to the 5 described in section 1.1.2. Note that by doing so, we guarantee smaller sets of data to be joined and we also benefit from parallelism. The next section elaborates on this new rule and the benefits it brings.

2.1.2. Reduction

After applying the equivalence rules, the syntactic optimizer goes through the reduction phase. After restructuring the trees by means of equivalence rules, this phase seeks to determine those subtrees that will produce empty relations (i.e., no data from that subtree is needed to answer the query). To do so, fragment expressions regarding the query definition are analyzed to find contradictions. Whenever a contradiction is detected (i.e., an empty result is detected), that fragment is automatically discarded.

As in the centralized counterpart, tautologies are detected and reduction techniques are applied to eliminate redundant fragment queries or empty results. However, in centralized environments tautologies are the result of mistakes when writing the query. In a distributed environment, they result from contradictions between fragment predicates and the query definition (thus, they do not entail any mistake in the query).

Reduction can happen on account of three specific equivalence rules:

- Union - Selection: Since set operations and selections commute, we will push down selection through unions, reconstructing a horizontally fragmented relation. In doing so, we reduce the amount of tuples each site has to ship to others. However, from the reduction phase point of view, you should note that if the selection predicate contradicts the predicate in the fragment definition, then that fragment does not need to be considered and can be removed.

Following the example previously introduced, since:

$$Q_1' := \sigma_{A>150}(R_1 \cup R_2 \cup R_3)$$

We can push the selection through the unions and get:

$$Q_1' := \sigma_{A>150}(R_1) \cup \sigma_{A>150}(R_2) \cup \sigma_{A>150}(R_3)$$

Now, substituting the definition of each fragment, we get:

$$Q_1' := \sigma_{A>150}(\sigma_{A<100}(R)) \cup \sigma_{A>150}(\sigma_{100\leq A\leq 200}(R)) \cup \sigma_{A>150}(\sigma_{A>200}(R))$$

Finally, fusing consecutive selections, the result is:

$$Q_1' := \sigma_{A>150\wedge A<100}(R) \cup \sigma_{A>150\wedge 100\leq A\leq 200}(R) \cup \sigma_{A>150\wedge A>200}(R)$$

Clearly, the result of the first selection is the empty set and the second and third ones can be simplified, resulting in:

$$Q_1' := \sigma_{150<A\leq 200}(R) \cup \sigma_{A>200}(R)$$

In terms of fragments:

$$Q_1' := \sigma_{A>150}(R_2) \cup R_3$$

This last step is performed by the reduction phase. As a consequence, some fragments are dropped and we only focus on those that are not contradictory to the query statement.

- Union - Join: Union and join commute by having the cross product of all possible joins. In this way, joins can be pushed down to the fragments, resulting in (many) more joins, with a view to exploiting parallelism (see section 2.2.2.) and, from the point of view of reduction, expecting some of them to be simplified. The reduction phase identifies useless (empty) joins by checking the query and fragment predicates to see if the join attribute defining the horizontal partition is the same in both relations.

We consider again the same fragments of R, and a new relation S with two fragments:

$$S_1 = \sigma_{A\leq 200}(S)$$

$$S_2 = \sigma_{A>200}(S)$$

In this case, our query is simply the natural join of these two relations:

$$Q_2 := R \bowtie S$$

First, we can substitute the fragments of each relation to obtain:

$$Q_2' := (R_1 \cup R_2 \cup R_3) \bowtie (S_1 \cup S_2)$$

We can push the join through unions by performing a cross product, and get:

$$Q_2' := (R_1 \bowtie S_1) \cup (R_2 \bowtie S_1) \cup (R_3 \bowtie S_1) \cup (R_1 \bowtie S_2) \cup (R_2 \bowtie S_2) \cup (R_3 \bowtie S_2)$$

Now, analyzing pairwise the definition of each fragment, we can see that out of six joins, only three of them can return tuples, because the predicates over A are contradictory for the other three, resulting in:

$$Q_2' := (R_1 \bowtie S_1) \cup (R_2 \bowtie S_1) \cup (R_3 \bowtie S_2)$$

Again, the reduction phase will drop some fragments and reduce the amount of work to be done.

- Projection - Join: Projection and join operations do not really commute, but we can still generate new projections under a join corresponding to the original projection. In this way, we also push projection down (through joins by reconstructing a vertical partition) to reduce the amount of attributes shipped by every site. You should note that if there are no common attributes (besides the beside key) between the query projection and the fragment definition, the fragment does not need to be accessed and thus, it can be removed in the reduction phase.

Let us consider now a fragmentation schema and the query below, over relation $T(\underline{A}, B, C)$ with primary key $\{A\}$:

$$T_1 = \pi_{A,B}(T)$$

$$T_2 = \pi_{A,C}(T)$$

$$Q_3 = \pi_{A,C}(T)$$

Next, we should substitute T with the join of these two partitions, resulting in:

$$Q_3' = \pi_{A,C}(T_1 \bowtie T_2)$$

We can push the selection through the join and get:

$$Q_3' := \pi_{A,C}(T_1) \bowtie \pi_{A,C}(T_2)$$

Now, by substituting the definition of each fragment, we get:

$$Q_3' := \pi_{A,C}(\pi_{A,B}(T)) \bowtie \pi_{A,C}(\pi_{A,C}(T))$$

Finally, if we simplify consecutive projections by performing the intersection of the projected sets of attributes, the result is:

$$Q_3' := \pi_A(T) \bowtie \pi_{A,C}(T)$$

Clearly, the result of this join is identical to that of the second projection, because each fragment has all the values of the primary key, resulting in:

$$Q_3' := \pi_{A,C}(T)$$

Which, in turn, can be represented in terms of fragments as:

$$Q_3' := T_2$$

2.1.3. Generating Alternative Trees

Remember that alternative trees are generated by applying the associative and commutative join rules previously introduced in section 1.1.2. There, we explained that centralized systems represent join trees as linear trees (either left or right-deep trees). However, distributed environments are not fond of linear trees. These trees boost pipelining, but they are more difficult to parallelize than bushy trees (in general, any join in a linear tree must continuously wait for the output produced by the previous join).

For this reason, distributed query optimizers better exploit bushy trees. As you will see in section 2.2.2., bushy trees (like the second tree in Figure 2) offer some benefits in terms of parallelism, because different branches can be executed completely in parallel without waiting for one another until they finish.

Bushy trees, however, generate many more options than linear trees, since it is no longer mandatory that one of the inputs be a physical relation. In fact, in our example of bushy tree in Figure 2, the join at the top has two intermediate results as input, but $A \bowtie B$ and $C \bowtie D$ can be completely executed in parallel. However, as has been said, the price to pay for such nice property may be too expensive, since producing all alternative trees by applying commutative and associative join rules yields a total of $\Theta(N!)$, where N is the number of relations to join. Indeed, the time to compute the alternatives could be worse than the actual execution time. For this reason, optimizers also exploit heuristics (such as focusing on relations with projections and selections first) when building bushy trees and, in this case, they are not as exhaustive as when building linear trees.

2.2. Physical Optimization

In a distributed query optimizer, not only must syntactic optimization be extended, but the physical query optimizer also has more work to do. Distributed systems distinguish between two kinds of physical optimization: (i) one that relates to physical structures (choosing the appropriate structures as well as their access paths) and maps to traditional physical optimization (see section 1.1.3.), and (ii) what is known as adding communication operators. The first one is called local physical optimization (and therefore, it is carried out individually at each site, querying the local catalog), whereas the second one, known as global query optimization, focuses on generating alternative process trees that, ideally, minimize shipping data over the network and benefit from parallelism as much as possible.

As previously discussed, in this module we focus on global query optimization, which is the new phase introduced in the physical optimizer. You can think of it as a global cost-based layer, which considers neither access paths nor physical structures (i.e., disk accesses, which will be addressed later by the local optimizer), but exclusively focuses on minimizing communication costs. In order to know which communication operators must be added to the process tree, we need to know how sites communicate among themselves. In other words, who does what and who sends data to whom. In this section we will focus on two main tasks which global query optimization is responsible for:

Syntactic vs. Physical optimization

Sometimes, the line between these two kinds of optimization is blurred. However, you can use the following rule of thumb to help distinguish between them: syntactic optimization aims to find the best operation order in the tree (or formulas), without considering physical structures or algorithms. By contrast, physical optimization is cost-based and needs this kind of information (i.e., physical structures available, communication costs, access paths, algorithms available, etc.).

- Producing the searching space: Unlike the local query optimizer, neither structures nor access paths are taken into consideration here. At this point, the main focus is how to efficiently execute joins by shipping less data.

- Benefiting from parallelism: Having *N* distributed database nodes, one of the main features of DDBMSs is parallelism. To this end, replication plays a major role, and the allocation schema is used at this point. The great news is that relational algebra has been proven to naturally fit parallelism, so no ad hoc parallel programming skills are needed.

As part of the physical optimizer, the global optimizer undertakes both tasks based on cost-based strategies. Unfortunately, this is a NP-hard problem. It is tackled by means of heuristics and dynamic rules, which tend to get close to the optimum solution in most cases. However, this is not always the case. To grasp the computationally expensive nature of this problem, consider the following scenario:

- A distributed database with 5 sites (i.e., database nodes): S_1, S_2, S_3, S_4 and S_5.
- 3 relations in the database R, S and T.
- Each relation is horizontally fragmented in two fragments (we refer to them by the name of the relation and a subindex, for example: R_1, R_2). You can consider them to be correct (i.e., complete, disjoint and reconstructible).
- Each fragment is replicated at all 5 sites.

Suppose now that the following query is issued in S_3: $Q_1 = \sigma(R) \bowtie \sigma(S) \bowtie T$ (selection and join attributes are not relevant for now), which would be translated into a fragment query as: $Q_1' = (\sigma(R_1) \cup \sigma(R_2)) \bowtie (\sigma(S_1) \cup \sigma(S_2)) \bowtie (\sigma(T_1) \cup \sigma(T_2))$. For such a simple setting realize that:

- Each selection can be individually executed at any site (i.e., 5 sites).
- The union between fragments of the same relation can also be made at 5 different sites. Thus, for each union, we have 25 alternatives for the selections combined with 5 choices to execute the union; i.e., 125 options per union.

- Next, we must consider combining all options for both unions – and yet, where (and how) to execute joins has not even been considered. In general, we would obtain 5^n alternatives (where n is the number of operations).

Clearly, this is an exponential problem, which cannot be exhaustively explored. However, you might be tempted to argue that, in this case, with all fragments replicated everywhere, executing the query in S_3 would be fairly reasonable. But suppose now that S_3 is two orders of magnitude slower than all the rest. Also suppose that these selections and joins are not very selective (i.e., thousands of hundreds of tuples are involved in this query) and executing all of it in S_3 (being I/O and CPU intensive) would be a bottleneck. Being replicated everywhere, we could easily benefit from parallelism by distributing tasks over all the sites. This is, indeed, the assumption made by many data-intensive systems that rely on the *divide-and-conquer* principle to deal with huge data loads.

Summing up, deciding where to execute what is a hard problem with lots of parameters to be considered: the communication cost throughout the network, cardinalities of intermediate results generated, processing capabilities of each site, etc. Furthermore, note that other non-functional requirements like data freshness, or site availability (generally known as constraints) could also add to the complexity of the problem. As a result, most current solutions just simplify the problem to consider the size of intermediate results and communication costs derived from them. Thus, solutions provided in these scenarios are also cost-based, like in centralized databases. However, the dominant factor here is bandwidth usage instead of disk accesses.

It is harder to have fresh data statistics in a distributed environment than in a centralized database. The global catalog can be replicated or stored in a single site or replicated in several sites, and it must store up-to-date statistics to allow the optimizer to do a good cost-based estimation when choosing an access plan. Gathering statistics from data stored in all sites (i.e., statistics about fragments and replicas) is not cheap. Furthermore, synchronizing the catalog copies might also be a problem if we decide to replicate it.

Note

Note that, nowadays, some networks are quite fast, and sending data over the network can be more efficient than performing disk accesses. You must be aware of your system settings to decide which kind of estimation better suits your system.

Subsequently, we divide this discussion into two main subsections. First, as you will see in Section 2.2.1., we discuss the search space that must be considered, focusing on how to deal with joins. Next, the search algorithm traversing that space must consider the degree of parallelism we support and take into account partitioning alternatives, as you will see in Section 2.2.2. In view of all this, it is easy to infer that the DBMS needs new heuristics to avoid a combinatorial explosion.

2.2.1. Generation of Execution Alternatives

In this section we focus on what alternatives we have to execute each piece of the query. Again, we focus on joins, the most expensive operation, which is what most DDBMs currently do. The syntactic optimizer deals with the problem of how to represent join trees and how to model and examine the search space in order to determine the optimum execution order. However, the syntactic optimizer is not aware of the presence of different sites. This constitutes another big challenge when dealing with joins in distributed environments, because different topologies of the syntactic tree (regarding sites) will also be generated.

Indeed, the problem is even bigger, since the DDBMS should decide how many sites are really necessary in the execution of each query. To consider all possible alternatives in the search space, however, is clearly unaffordable, as it already was in the simplest case of centralized systems. Thus, it is usually assumed that all available processing sites are used for every query.

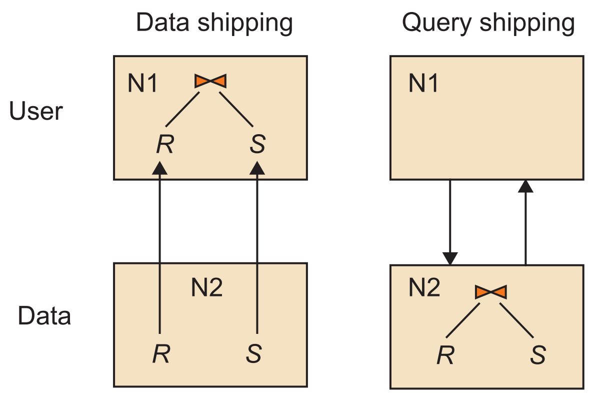
At this point, we must decide where to execute each join. Note that this is a crucial issue, because necessary data must be shipped to the chosen site, and a fair amount of data may be involved. For this reason, a new operator is introduced in the process tree, called *exchange*[2], which shows data shipped from one site to another. To simplify the explanation, let us assume that we only have two sites, one containing the data and another where the user issues the query. An example scenario would be that of figure 5, which contains two sites (namely, N_1 and N_2). Suppose now that a query containing $R \bowtie S$ is issued in N_1, but these two relations (neither fragmented nor replicated) are stored in N_2. In this case, we have two possibilities:

(2)In general, the exchange operator can represent data shipped by any operation, but remember we are focusing on joins in this entire section.

1) Data shipping: The data is retrieved from the stored site to the site executing the query. This avoids bottlenecks on frequently used data, as the stored site only provides data but does not execute any operation (i.e., left side of figure 5).

2) Query shipping: The evaluation of the query is delegated to the site where the data is stored. Conversely, this avoids transferring large amounts of data (i.e., right side of figure 5).

Figure 5. Shipping possibilities with only two sites



Ideally, the best solution is a hybrid shipping strategy, which finds a balance between transferring data and executing operations in a frequently-queried site. Again, the same problem pops up: to analyze the total number of different alternatives, depending on the size of the inputs and outputs of every operation in the process tree, is simply unaffordable.

Multi-way joins

Regarding data shipping for joins, we consider only joins between two tables. The problem is more difficult for multi-way joins that operate on more than two tables at the same time.

Alternatively, this hybrid solution has been implemented by means of semi-join reductions. This strategy has been widely accepted in distributed environments and has proved to be of great value in decreasing the amount of data sent over the network. A semi-join is a typical join operation, with the only difference being that the output contains the attributes of one input relation (either left or right). Formally, where A_i are the attributes of R, the left semi-join between R and S is $R \ltimes S = \pi_{A_i}(R \bowtie S)$. Symmetrically, where B_i are the attributes of S, the right semi-join between R and S is $R \rtimes S = \pi_{B_i}(R \bowtie S)$.

How can semi-joins help in minimizing the amount of data shipped between sites? When two tables (R and S) that need to be joined reside in different sites (N_1 and N_2, respectively) we obviously have the possibility of sending R from N_1 to N_2 or S from N_2 to N_1. However, it is less obvious that there are also two more possibilities: sending only the join attributes. That is, sending the join attributes of R from N_1 to N_2, perform a semi-join at N_2 (i.e., $S \ltimes R$) and send the result back to complete the join at N_1. Symmetrically, the DDBMS could also choose to send the join attributes of S to N_1 and proceed similarly. Essentially, note that the following equivalence is being exploited: $R \bowtie S = R \bowtie (R \rtimes S) = S \bowtie (R \ltimes S)$. Intuitively, with the first semi-join we know which tuples of S can participate in the join and we can ignore the rest. These tuples, and only these that participate in the join, are sent to N_1, where by definition, all tuples coming from N_2 will join (in other words, there are no dangling tuples).

The benefit of such an approach is what we have previously discussed: it is a hybrid approach that combines data and query shipping. Most importantly, it reduces the communication overhead, because only join attributes are sent first, and then only those tuples known to be in the result eventually follow.

Note

Moreover, note that we even have the possibility of performing two semi-joins and join both results in a third site $R \bowtie S = (R \ltimes S) \bowtie (R \rtimes S)$.

As a negative aspect, note that we are performing more operations. Thus, as general rule, a semi-join should be considered if we have a small join selectivity factor (i.e., the result of the operation is a small percentage of the input tables). However, again, the statistics needed to make such a decision might not be available.

Note

In fact, when sending tuples in the second step, the DDBMS does not even need to send the complete tuples, but just projections of them containing those attributes needed to answer the current query.

In general, in order to decide whether it is worthwhile to use a semi-join strategy, we should at least check that the size of the relation to be shipped (S) is bigger than the projection of the join attributes to be sent from N_1 to N_2 plus the result of the semi-join($S \ltimes R$).

2.2.2. Parallelism

In a distributed system it is important to employ parallel hardware effectively. Essentially, parallelism is obtained by processing different pieces of data in different processors (note that in the context of DDBs, these processors reside in different sites).

Thus, serial algorithms need to be adapted to multi-thread environments, where the input data set is divided into disjoint subsets. On the one hand, this raises a potential problem about concurrency and contention conflicts when accessing resources and, furthermore, it may negatively impact the overall execution time (i.e., throughput) if we add the processor time consumed at all sites. On the other hand, it is expected to reduce the response time (at least, when dealing with very large databases) and provide scalability and high-availability.

Ideally, parallel programs should pursue linear speed-up and linear scale-up. Both speed-up and scale-up are scalability measures. Speed-up, also known as horizontal scalability, measures performance when adding hardware for a constant problem size. A claim of linear speed-up means that adding computing power (e.g., new sites) should yield a proportional increase in performance (i.e., N sites should solve the problem in $1/N$ time). As for scale-up, also known as vertical scalability, this measures performance when we consider that the problem size is altered with resources. Linear scale-up refers to sustained performance for a linear increase in both database size and number of sites (i.e., N sites should solve a problem N times bigger in the same time).

Fortunately, relational algebra naturally benefits from parallelism in the presence of fragmentation and/or replication. Thus, no specific parallel algorithms are needed. This is a huge advantage, since parallel programming skills are known to be a bottleneck in many scenarios.

There are different alternatives for a DDBMS to benefit from parallelism – namely, inter-query parallelism and intra-query parallelism. We talk about inter-query parallelism when several queries are executed in parallel. However, this is easily achieved by all DBMSs that support concurrency. Thus, in this section we are focusing on intra-query parallelism, where different parts of the same query are executed in parallel.

For intra-query parallelism, we have, in turn, two alternatives:

- Inter-operator: several nodes of the same process tree are executed in parallel. For example, two selections over two different relations (e.g., *R* and *S*) that reside in different sites, can be executed in parallel, benefiting from distribution.

- Intra-operator: several parts of the same node in the process tree are executed in parallel. For example, a selection on a fragmented relation (e.g., *R*, which is fragmented into two fragments; R_1 and R_2) can be executed in parallel by selecting on different fragments at each site and eventually uniting the result obtained from each of them.

Nowadays, the main need for distributed processing arises from the current role played by very large databases. Indeed, distributed processing is conceived as the best way to handle large-scale data management problems, in that it can be considered to be a straightforward application of the divide-and-conquer approach. If the necessary software and hardware needed for a very large application is beyond current capabilities of centralized systems, distributed systems can provide the solution by breaking up the problem into small logical pieces, and working in a distributed manner.

1) Inter-operator parallelism

Two different possibilities are available for inter-operator parallelism, depending on the topology of the process tree. If it is a bushy tree (note that, in this section, we talk about bushy trees in general, either join trees or trees involving any other *n*-ary operation), independent branches can be easily executed in parallel. On the other hand, with linear (right or left-deep) trees, even if tuples are pipelined from one operator to the next, all (or some) of these nodes can still work in parallel (i.e., one site can process one tuple while another site or processor is processing a different tuple). However, parallelism in linear trees is trickier, as different strategies must be applied depending on the operator (in some cases, it may not even be worth it) and a final merge phase might be needed.

One of the most used strategies to incorporate parallelism is the use of buffers. In this case, the producer leaves its result in an intermediate buffer and the consumer retrieves that content asynchronously. In this way, the producer can eagerly generate output tuples until the buffer becomes full. By using buffers, though, we risk stalling the system. Stalls happen when an operator becomes ready and no new input is available in its input buffer. Note that this stalling scenario is propagated like a bubble through the buffer chain (i.e., if the input buffer of one operator becomes empty, it will stop generating tuples and its output buffer will soon become empty, affecting subsequent operations in the chain).

To better visualize the problem, let us define *latency* as the time needed to get the result of the query, and *occupancy* as the time occupied until the DBMS can accept more work. In other words, latency tells us the amount of time needed to *answer* the query (i.e., from when the query is issued until the query is answered), whereas occupancy tell us the time a component needs to accept a new query (i.e., the time needed by every component to execute every single query piece). Assuming that we have N operators to parallelize, h is the height of the tree (either a linear or bushy tree), T time units are required for the whole query and k is the delay imposed by imbalance in case of stall, Figure 6 shows the values of these two measures:

Figure 6. Comparison of latency and occupancy in a stalling system

		Latency	Occupancy
Serial system		T	T
Parallel	No stalls	$h \cdot T/N$	T/N
	Stalls	$h \cdot T/N + h \cdot k$	$T/N + k$

Note

Remember that, by definition, each operation described in a linear tree always takes at least one relation as input. Consequently, the height is always equal to the number of operations to parallelize.

Importantly, note the benefit of having bushy trees instead of linear trees. For linear trees, in these formulas, N is always equal to h. Refer again to figure 2. There, both trees have the same amount of operators to parallelize (i.e., $N = 3$). However, the left-deep tree has a height of 3 (note that we consider the root to be height 0), while the bushy tree has a height of 2. In general, the larger the number of operations to parallelize, the bigger the difference in height between a left-deep (or right-deep) tree and its bushy counterpart. Consequently, even though bushy trees can also suffer from stalling, they are easier to parallelize and, in the general case, they better exploit parallelism.

2) Intra-operator parallelism

Intra-operator parallelism is always based on fragmenting data. Since query processing manipulates sets of tuples, these sets can be splited into disjoint subsets, so that each result is generated only once and eventually merged. In the case of manual fragmentation, like that of homogeneous distributed systems, we can benefit from intra-operator parallelism. However, if the relation is not fragmented, the DDBMS can fragment it on-the-fly in order to gain this benefit. Specifically, there are different options for this kind of dynamic fragmentation:

- Random or round-robin: This strategy balances the load but blinds the searches.

- Range: This option facilitates directed searches, but needs accurate quartile information.

- Hash: Hash strategies facilitate directed searches, but depend dramatically on the hash function we choose. These will always have problems in the presence of non-unique attributes with many repetitions.

In the case of dynamic fragmentation, a new fragmentation property has to be added to each operator in the process tree. This property must contain information about the fragmentation strategy being used (i.e., random, hash, etc.), the specific fragmentation predicates used and the number of fragments produced.

Parallel databases

Parallel databases take intra-operator parallelism to the extreme. The input(s) of an operation is (are) dynamically (i.e., at execution time) partitioned and parallelized. Thus, no fragmentation is required beforehand.

For example, if we want to join *R* and *S*, we can fragment *S* into S_1 and S_2 and now, by replicating *R* in two different sites and distributing the two fragments of *S*, we can parallelize it as $R \bowtie S_1$ and $R \bowtie S_2$. Even more, if *R* is also fragmented by means of the corresponding join attribute (i.e., we obtain R_1 and R_2), we do not need to replicate anything, but ship half the relation to the corresponding site, because $R \bowtie S = (R_1 \bowtie S_1) \cup (R_2 \bowtie S_2)$. Thus, we not only gain parallelism, but also efficiency from the point of view that we expect those two joins to be cheaper than a unique join.

Note

Remember that we distinguish between fragmentation and partitioning. Fragmentation is the problem of breaking a relation into smaller pieces to be distributed over a network. Partitioning is essentially the same, but the pieces produced are kept locally. Partitioning is typically related to parallel databases, whereas fragmentation is related to DDBMS.

In general, for binary nodes of the process tree (such as join or set operators), the fragments can be generated dynamically (i.e., at query time). Moreover, only the attributes available for that operator are considered as candidates to generate the fragmentation predicates. For example, consider the following scenario:

- R(rid, a_1, a_2),

- S(sid, rrid, b_1, b_2) where rrid is a FK to R(rid),

- and the following process tree: $\left(\sigma_{rid,a_1}(R)\right) \bowtie_{rid=rrid} \left(\sigma_{rrid,b_2}(S)\right)$.

In this case, if we want to parallelize the join (note it is the only binary node in the process tree), the DDBMS can dynamically fragment the join inputs and distribute them. Typically, the fragmentation strategies applied by DDBMS are hash or round-robin strategies, which can be easily applied without human interaction (i.e., using a predefined hash function or randomly distributing tuples to the available sites).

However, note that the DDBMS must pick an attribute to be used in the fragment predicates. In our example, there are only two candidates for each join input: *rid* or a_1 for the first one (note that a_2 cannot be used), and *rrid* and b_2 (i.e., *sid* and b_1 cannot be used) for the second one. If the database had two different sites, the DDBMS might decide to use a hash function (for example, in case of numbers, checking if they are even or odd) over *rid* and *rrid* and accordingly, distribute both inputs. Even values would be sent to one site, and odd numbers to the other one. Now, the DDBMS can perform the join in parallel, because matching values from both inputs will be sitting in the same site.

As for unary nodes of the process tree (e.g., projection or selection), they can only benefit from intra-operator parallelism in the presence of static fragmentation (in the sense that the fragmentation strategy must have been provided in advance; i.e., at design time and therefore, is only profitable for homogeneous distributed systems). A typical strategy to fragment relations and benefit from parallelism in the presence of unary nodes is to store data according to small value ranges among the available storage sites (even if the fragments produced do not really make sense from a conceptual point of view).

For example, consider the scenario just presented above. Suppose now that the attribute a_1 is a number representing the age of people. We also know that this attribute is frequently queried in our database. In this case, the DBA could decide to fragment R with regard to a_1. Two possible alternatives to be considered by the DBA are to statically define the ranges (e.g., 1-18, 19-30, 31-45, 46-65 and 66-99) or use a hash function (e.g., using the modulo operation). According to the range produced by the chosen strategy, the tuple will be stored in a specific site. For example, if 5 sites are available, each of the 5 static ranges would be placed in a different site. If hash were chosen, a modulo 5 function would be a good option.

In this way, small range queries (not worth being parallelized) will be executed in just one site, while large queries involving large ranges spread over several sites are executed in parallel, balancing the load among the different sites. For example, assume the static range fragmentation strategy proposed. Queries containing selections such as $a_1 = x$ where x is a number between 1 and 99 will be processed in just one site. Queries involving large ranges, such as $a_1 <> x$ would be executed in parallel, as it involves checking almost all the tuples. To do so, the DBMS will take advantage of all 5 fragments to parallelize such operations and eventually unite the results produced at each site.

As an extension to this strategy, hybrid fragmentation strategies are also possible. For example, one column can be hashed in order to determine the site where the tuple should be stored, and a different column is used then to decide on the storage device inside the site. However, note that using the same hash function at both levels is a bad idea, since tuples assigned to the same site are guaranteed to produce the same hash value (and thus, all of them would be assigned to the same device).

2.2.3. Access Plan Evaluation

At the end of the physical optimization process, we will obtain a set of alternative process trees (each one representing an access plan), for each alternative tree generated during the syntactic optimization process. Thus, we must decide which of these process trees is the most efficient and, to do so, we need a cost model that quantifies how expensive each strategy is. Typically, cost of an access plan is computed in terms of latency, as the sum of local costs (provided by the local optimizer) plus communication cost. All in all, we should gather information about:

Note

In this section we distinguish between a tuple and an instance. A tuple is physically stored in a relation, whereas an instance is every single row produced by an operation; thus, an instance may be a tuple itself, but may also be the result of projecting some attributes from a tuple, joining two tuples, etc.

- Local processing:
  - Average CPU time to process an instance (T_{CPU})
  - Number of instances processed (*#inst*)
  - I/O time per operation ($T_{I/O}$)
  - Number of I/O operations (*#I/Os*)
- Global processing:
  - Message time (T_{Msg})
  - Number of messages issued (*#msgs*)
  - Transfer time (to send a byte from one location to another) (T_{TR})
  - Number of bytes transferred (*#bytes*, but it could also be expressed in terms of packets)

The first set of items measures local processing time. Mainly, local cost has to take into account the cost of central unit processing (i.e., number of clock cycles needed), as well as that of disk I/O (i.e., number of transfer operations to/from disk). The second set of items measures global processing time (i.e., communication costs). Thus, it considers the time to initiate/send messages (e.g., for synchronization purposes between sites) and the time sending data. Accordingly, the total time could be expressed as:

$$RESOURCES = T_{CPU} * \#\, inst + T_{I/O} * \#\, I/Os + T_{Msg} * \#\, msgs + T_{TR} * \#\, bytes.$$

Note

The total cost is normally expressed in time units.

All in all, the knowledge we require to compute this cost, as in the centralized case, is basically related to the quantity of data and statistics needed to obtain the selectivity factor of operations. Remember that selectivity factors are needed to estimate the size of intermediary results. The main problem with this approach it that it does not take into account the usage of parallelism, where operations are divided into *N* pieces. Alternatively, to accommodate parallelism in our formulas, we can use a cost model expressed in terms of the actual response time (and not in terms of resources consumed to execute the query). For example:

$$Response\_time = T_{CPU} * seq_{\#insts} + T_{I/O} * seq_{\#I/Os} + T_{msg} * seq_{\#msgs} + T_{TR} * seq_{\#bytes}$$

Where $seq_{\#x}$ is the number of x (x being either instances, I/O operations, messages or bytes) that must be done sequentially for the execution of the query. Among them, parallel processing is not considered.

Finally, note that many simplifications or assumptions are made in real cost models. For example, even distribution of data is assumed or, commonly, communication cost between sites (i.e., T_{TR}) is assumed to be constant. However, this could not be true for WANs, and leads to skewed results, but it drastically simplifies the problem.

Cost models can be expressed with respect to either resources consumed on response time. Be aware of the consequences of each approach.

Summary

In this module, we have introduced query optimization for distributed systems. The input query is supposed to be an SQL statement and the output is an access plan (in a given procedural language). In general, this access plan minimizes a set of parameters expressed in a cost model used to decide among alternative access plans considered during the optimization process.

Query optimization passes through three main stages: semantic, syntactic and physical optimization. Semantic optimization rewrites the SQL query into an equivalent, more efficient SQL query. The syntactic optimization describes the SQL query in terms of relational algebra and looks for an order of operations that minimizes the size of data flow. Finally, the physical optimizer takes into account physical structures, access paths and algorithms that implement the operations, in order to decide, based on a cost model, which access plan should be proposed. In a distributed environment, all these stages pursue the same goals as in a centralized system, but they require extension in order to account for distribution. Thus, syntactic optimization is extended to locate fragments (a reduction phase is also added to identify only those fragments of interest to answer the query), and physical optimization is extended with a global step where communication costs are accounted for.

In practice, the inherent complexity of distributed query processing forces current DDBMSs to work on the basis of certain assumptions that simplify the problem.

Glossary

Allocation (or data allocation) This problem appears in top-down design of distributed databases. Given a set of fragments, the data allocation problem aims to allocate them to the available sites in such a way that certain optimization criteria are met.

ANSI/SPARC architecture The reference schema architecture for DBMSs

CNF Conjunctive Normal Form

DAG Directed Acyclic Graph

DB Database

DBA Database Administrator

Data locality Related to distributed systems, it refers to the act of placing data where it is needed to minimize communication overhead and, consequently, be more efficient and achieve better performance

DDB Distributed Database

DBMS Database Management System

DDBMS Distributed Database Management System

FK Foreign Key

Fragmentation The problem of breaking a relation into smaller pieces to be distributed over a network

Fragmentation predicates A set of selections that describe a horizontal fragmentation

LAN Local area network

Latency Time to process a query

Occupancy Time until a DBMS can accept more work

Partitioning Essentially, it follows the same principle of fragmentation but, does not spread resulting fragments over a network, keeping them instead in local. Partitioning can be used for many purposes, but it is mainly used to benefit from parallelism and to implement privacy

PK Primary Key

Replication When the same fragment is allocated in several sites. It is mainly used to improve reliability and efficiency of read-only queries

Scalability In distributed systems, it refers to the system expansion

Speed-up This is a measure of scalability where we consider additional hardware for a constant problem size

Scale-up This is a measure of scalability where we consider that the problem size is altered with the resources

SQL Structured Query Language

Transparency This refers to separation of the higher-level semantics of a system from lower-level implementation issues

WAN Wide area network

Bibliography

Liu, L.; Özsu, M. T. (Eds.) (2009). *Encyclopedia of Database Systems*. Springer.

Özsu, M. T.; Valduriez, P. (2011). *Principles of Distributed Database Systems*. 3rd Edition, Prentice Hall.

Ramakrishnan, R.; Gehrke, J. (2003). *Database Management Systems*. 3rd Edition, McGraw-Hill.

Graefe, G. (1993). *Query Evaluation Techniques*. In ACM Computing Surveys, 25(1), June.

Ioannidis, Y. (1996). *Query Optimization*. In ACM Computing Surveys, vol. 28, num. 1, March.