2021

(2021 Revision) Chapter 5: Essential Aspects of Physical Design and Implementation of Relational Databases

Tatiana Malyuta
Tatyana#09

Ashwin Satyanarayana
CUNY New York City College of Technology

Chapter 5. Query Processing and Performance

The users of database applications have definite expectations about the time it will take the database to respond to their requests or the number of requests the database will be able to process per unit of time. The user requirements on performance must be met by the database and database applications, and often the success of the database is defined by its performance.

This chapter is about a special relationship between the database and database applications. Though very often we say 'database performance', actually, we mean the performance of the database applications. However, good performance of applications can be achieved only on appropriately designed and implemented databases. On the one hand, databases have to be designed and implemented with the aim of satisfying the needs of specific database applications. On the other hand, the applications have to be designed and implemented with regard to the database solution and the features of the DBMS that are responsible for the processing of the application's requests to the database.

In this chapter we will discuss requests to the database that result in retrieving and modifying queries. Improving the performance of query processing, also called *database tuning*, is one of the most important problems of database design and implementation. It is also one of the most complicated problems to solve because performance depends on many factors. This chapter discusses the general problems of query processing, the features of DBMSs that allow for improving performance, and the aspects of the database and database applications that have an impact on query processing and performance.

Problems of Query Processing

Goals of Query Processing

The effectiveness of query processing is usually measured with the help of the response time, throughput, and total time or cost. The objectives of query processing are minimizing the total cost and response time, and maximizing the throughput.

The *total cost (total time)* is the sum of all the times spent for processing the operations of a query. It includes the cost (time) of the Input/Output (I/O) and CPU operations. The I/O cost is the time spent on disk reading/writing operations: the system searches the disk for data, reads data blocks into memory and writes data back to disk. The CPU time is the time spent on data processing in memory (RAM).

Total Cost =
(Number of I/O operations * Cost of one I/O operation) +
(Number of RAM operations * Cost of one RAM operation).

The cost of disk access is significantly higher than the cost of memory operations, therefore, to improve performance, it is usually important to minimize the number of disk accesses. In a distributed database, the total cost of distributed queries also includes the cost of data transfer between the databases.

The *response time* is the elapsed time for the execution of the query. The response time is composed of the actual query processing time and the waiting time, when the system waits for the needed resources to become available.

Response time = Processing time + Waiting time.

If there is no resource contention (and the waiting time is zero or insignificantly small), then the response time can be less than the total time (total cost) because some operations of a query can be processed by the system in parallel. For example, imagine that three operations are involved in the processing of a query: the first operation requires 0.1 units of time, the second – 0.3 units, and the third – 0.6 units. The total time for processing this query is 1 unit of time (0.1+0.3+0.6). If the operations are executed simultaneously, then the response time is 0.6 units. However, if some of the needed resources are unavailable, for example locked by another user (see Chapter 6 on transaction management), then the response time can be significantly higher.

The *throughput* is the number of queries[1] processed per unit of time. The throughput can be increased by reducing the total or processing times, or by eliminating resource contention and reducing the waiting time.

Throughput = Number of executed queries per unit of time.

Different database applications have different processing goals. For example, the On-Line Transaction Processing (OLTP) applications have to perform many data requests in a limited time, and, therefore, their primary goal is to maximize the throughput. The On-Line Analytic Applications (OLAP), on the other hand, are characterized by a small number of concurrent requests. However, the requests are often complicated and resource consuming, and minimizing their response time is important. Figure 1 schematically illustrates how with the increase of the number of concurrent users the response time is getting worse while the throughput is getting better (until some critical number of concurrent sessions is reached).

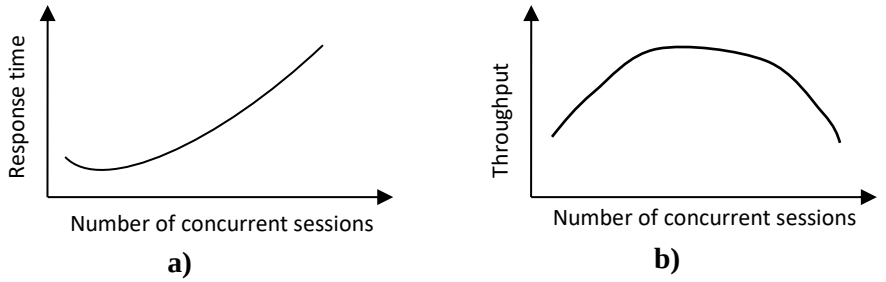

Figure 1. Response time vs.throughput.

This chapter discusses how to achieve better performance by minimizing the processing and response times and increasing the throughput, assuming that there is no resource contention and the system is not waiting for resources to become available. Chapter 6 explains resource contention problems in the multi-user environment and approaches to improve performance of concurrent queries by minimizing the waiting time.

Query Decomposition

Database requests are written in a high-level language; for relational databases it is the SQL dialect of a particular DBMS. Query processing starts with the decomposition or transformation of the query into a sequence of operations of relational algebra (relational operations are explained in Appendix 2). The sequence of relational operations defines the strategy of the query execution. Most SQL queries can be transformed

[1] In reality, it is the numer of transactions which are discussed in Chapter 6.

into more than one relational expression, and the purpose of query decomposition is to choose the expression that gives the best (or reasonably good) performance.

The discussion of query processing will be based on the Manufacturing Company case. Consider the query that finds data about employees who work in the IT departments:

```
SELECT e.*, d.*
FROM Employee e, Department d
WHERE e.deptCode = d.deptCode AND deptType = 'IT';
```

This query can be decomposed into two different relational expressions:

1. First find the Cartesian product of the tables Employee and Department, then select from the result the rows that satisfy the condition of the join of these tables, and then select the rows of the IT departments:

$$\sigma_{\text{Department.deptType='IT'}}(\sigma_{\text{Employee.deptCode = Department.deptCode}}(\text{Employee x Department}))$$

2. First select all rows from the table Department with deptType = 'IT' and then join the result with the table Employee by deptCode:

$$\text{Employee} \blacktriangleright\blacktriangleleft_{\text{deptCode}} \sigma_{\text{deptType='IT'}}(\text{Department})$$

Query Optimization

The second relational expression looks simpler than the first one. The following rough estimation of the processing costs for both expressions demonstrates that choosing a good execution strategy is important for good performance. To simplify the estimation we will make the following assumptions:

- The results of intermediate processing are stored on disk. Later in this chapter it is shown that the results of the intermediate operations often are kept in memory – this allows for reducing the number of I/O accesses and, therefore, improving performance.
- Rows of the tables and of intermediate results are accessed one at a time. In reality, as discussed in Chapter 2, data is read by data blocks that contain multiple rows. The number of rows in the block depends on the rows' length, the percent of free space in the block, and the size of the block (the block size is defined by the administrator to accommodate the needs of applications in the optimal manner).

The estimation is based on 200 departments, with 40 IT type departments. 4000 employees are distributed evenly across the departments – approximately 20 employees in a department.

Because the cost of memory operations is much less than the cost of I/O operations, the cost of a strategy is estimated as the number of I/O operations.

For the first strategy:

1. For the Cartesian product, read all the rows of the tables Employee and Department: 200 + 4000 accesses. The Cartesian product results in 4000 * 200 rows.
2. For the intermediate result of the Cartesian product, write all rows to disk: 4000 * 200 accesses.
3. For the first selection (join), read the intermediate result of the Cartesian product: 4000 * 200 accesses. The first selection results in 4000 rows.
4. For the intermediate result of the first selection, write all rows to disk: 4000 accesses.
5. For the second selection (deptType = IT), read the rows of the previous result: 4000 accesses.

The cost of the first strategy is:

$$4200 + 4000 * 200 + 4000 * 200 + 4000 + 4000 = 1612200.$$

For the second strategy:

1. For the selection (deptType = IT), read all rows of the table Department: 200 accesses. The selection results in 40 rows.
2. For the intermediate result of the selection, write all rows to disk: 40 accesses.
3. For the join, read all rows of the table Employee and rows of the intermediate result: 4000 + 40 accesses.

The cost of the second strategy is:

$$200 + 40 + 4000 + 40 = 4280.$$

The difference between the processing costs of these two relational expressions (and, correspondingly, performances of the query) is striking. Choosing a good strategy is important for performance. Defining an efficient query execution strategy is called *Query Optimization*. Optimization plays an important role in query processing. The efficient decomposition of a query is only a part of the query optimization process. The following sections demonstrate how the database optimizer chooses between different ways of accessing and processing data for a given sequence of operations produced by the decomposition.

General Scheme of Query Processing

Figure 2 shows the main steps of query processing.

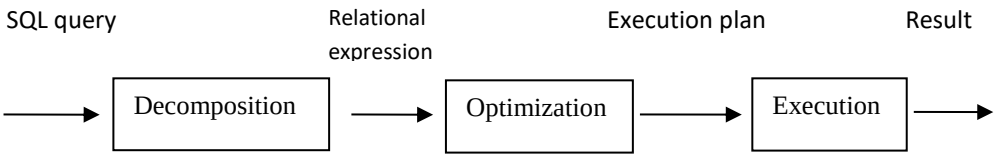

Figure 2. Scheme of query processing.

In the previous section it was shown how the query is decomposed into relational operations. In the beginning of the decomposition, the query is analyzed for correctness. First, the query must have the correct syntax; the processing of queries with syntax errors is cancelled. The decomposition step also includes analysis of the query: checking that the references to the objects of the database are correct and that the user who started the query is authorized to reference these objects (tables, views, synonyms, etc.). If the user is not authorized to access at least one of the objects of the query, the processing is cancelled.

The analysis of the query may include limited checking of the query's logic. For example, if a query condition includes the following expression:

```
WHERE … AND deptType = 'IT' AND deptType = 'IT',
```

it will be simplified into:

```
WHERE … AND deptType = 'IT'.
```

However, the programmer should not rely on the query processing capabilities of the database for improving the logic of the query. If, for example, the join conditions for the tables of a query are not specified, most DBMSs will not react to this and the query will return the Cartesian product of the tables. The main responsibility for reasonable and logically correct queries is on the programmer's shoulders. Some basic recommendations for composing "good" queries are discussed later in this chapter. In general, query processing is performed for any syntactically correct query with authorized access to all required resources. The programmer is responsible for making the query meaningful.

Types of Optimization

The optimization of a query starts with its decomposition, when the system tries to build a relational expression implementing an effective sequence of relational operations. The optimization of a query does not end there. For each relational operation in the sequence, there are different ways to search and read the necessary data, and to process the data in memory. Today's DBMSs offer two basic types of optimization: heuristic and cost-based.

Heuristic Optimization

Relational operations have different processing complexity (see Appendix 2). Systems with a heuristic (also called rule-based) approach to optimization use heuristic rules based on the complexity of the relational operations and the transformation rules for relational operations. The detailed description of transformation rules can be found in [Őzsu]. By applying these rules, the system tries to rebuild the relational expression and make it more efficient. For the above example of query decomposition, the optimizer will choose the more efficient relational expression, which in this example happens to be the second expression.

Query optimization does not end with decomposition. After decomposing the query, the system applies definite rules for choosing the strategy of accessing the data. For example, for the discussed query the system has to read rows of the IT departments (deptType=IT) of the table Department. If there is an index on the deptType column of the table, the system will choose indexed access to the data of the table because according to heuristic rules indexed data access is preferable to full table access. Heuristic rules are formulated for the most common situations and do not take into consideration a particular state of the database, that is why they do not necessarily give the best strategy for each particular state of the database. Heuristic optimization was used in the earlier versions of DBMSs.

> **Heuristic optimization is based on the rules of complexity and the transformational rules of relational operations.**

Some of the basic heuristic rules are:

- Perform the selection operations as early as possible. In the discussed example, the second relational expression, which started with the selection on deptType, was more efficient because fewer rows were passed to subsequent operations.
- Transform a Cartesian product that has a subsequent selection based on a join condition into the join operation. In the first discussed strategy, the Cartesian product of Employee and Department was followed by the selection with a join condition between these two tables. In the second strategy, which was more efficient, in place of the Cartesian product and the subsequent selection we used a join (recall the definition of the join operation from Appendix 2).

- Execute the most restrictive selections first – performance is better if fewer rows go to subsequent operations, e.g. on our example. first select all rows from the table Department with deptType = 'IT' and then join the result with the table Employee by deptCode:

$$\text{Employee} \blacktriangleright\blacktriangleleft_{\text{deptCode}} \sigma_{\text{deptType='IT'}}(\text{Department})$$

Cost-Based Optimization

Another way to find the optimal strategy for carrying out the query is to estimate the costs of the different strategies and choose the strategy with the minimum cost, similar to the approach shown earlier in this chapter. The estimation of the cost requires database statistics which are measurements of the physical properties of the database. In the above example, the estimation of strategies was based on information about the number of rows in the tables and the number of rows for the IT departments. The accuracy of cost-based optimization depends on how detailed and up-to-date the database statistics are.

The database statistics are usually kept in the database dictionary and include:

- Cardinality (number of rows) of tables.
- Number of data blocks (pages) occupied by tables.
- Number of rows per block for tables.
- Number of distinct values in columns of tables.
- Maximum and minimum values of columns of tables.

> **Cost-based optimization estimates costs of strategies with the help of the database statistics.**

Keeping the statistics updated after every modification of data is not feasible because statistics recalculations and updates in the data dictionary create processing overhead and decrease performance. That is why statistics are gathered on a periodic basis depending on modifying activities in the database, e.g. several times a day for a database with intensive modifying activities, and less frequently for a database in which data are modified less often.

Static and Dynamic Optimization

There are two ways to build and execute a strategy – to accept the strategy developed *before execution* and follow it, or to start with the developed strategy, while re-evaluating and changing it *during execution* if the initial steps do not give the expected result. The first approach is called static optimization, and the second – dynamic optimization. Most DBMSs use static optimization, though there are significant efforts being performed to produce efficient dynamic query optimization; later in the chapter we briefly discuss Adaptive Plans of Oracle.

Factors That Influence Performance

Though there was tremendous progress in query processing and query optimization capabilities of DBMSs, the performance of database applications is defined not only by the power of the query optimizer but also by many other factors.

The design of data plays an important role in the performance of the future database. The design of data has to correspond to the nature of the database applications, e.g. for OLTP databases the data model is usually normalized, while for OLAP databases, the data model is denormalized.

The implementation of the database, including the organization of data storage, has a major impact on performance. The database has to be designed and implemented considering the expected reading and modifying activities, volumes of data, number of concurrent users, and other factors.

The design and implementation of the database application have to be performed with the understanding of the database design and implementation, and the features of the DBMS. Ignoring specifics of the database implementation and not utilizing the DBMS features in the application can cause unsatisfactory performance.

The performance of the database depends upon the database environment (the hardware and software) and the degree to which the database implementation utilizes this environment.

This chapter concentrates on database-related factors that influence performance and features of the DBMS, which can be used for performance improvement. General recommendations about application design are discussed in this chapter and are discussed in more detail in Chapter 6 on transaction management.

Database Design

When considering a design solution for a database, it is important to remember that a particular design is rarely equally beneficial for all database requests. Therefore, the designer's efforts must be directed on achieving the best performance for the most important and frequent queries.

For OLTP databases, the primary goal of the database design methodology is to build a normalized database free of update anomalies. The performance of modifying operations benefits from the normalized design because each fact is stored in the database only once and if it is modified, the modification takes place once. However, the processing of read queries in the normalized database is often complicated because of numerous tables (compared to denormalized databases) and the necessity to join data from multiple tables in queries. The join is an expensive operation in terms of processing, and adding a table to a join query can cause performance problems. For example, if the table Title is added to the join of the query about employees of the IT departments, performance becomes several times worse.

The performance of read queries can be improved by denormalization of the database. Suppose that queries about employees of departments of different types for the Manufacturing Company database are executed often and their performance is important. Denormalization of the database by merging the tables Department and Employee can be beneficial for the performance of these queries. Denormalization results in the new table Department_Employee (deptCode, deptName, location, deptType, ID, emplName, emplType, deptCode, titleCode), which contains data about employees and the departments they are assigned to. Let us estimate the performance of the query about employees of the IT departments on the new table. The table contains 4000 rows (because one employee works in one department), and the cost of retrieving the ID and name of employees of the IT departments is the cost of reading all the rows of the new table: 4000. Denormalization improved the performance of the query.

The performance of read queries that involve multiple large tables can be dramatically improved by denormalizing the database. Note that the improvement of performance of read queries comes at the cost of performance of some update queries. For example, updating a department's name, which in the normalized database requires changing one row, causes changes in approximately 20 rows of our denormalized database. Besides, the denormalized database needs additional measures to support data consistency – in our case, it is important to ensure that the data about every department are consistently supported across multiple rows with the department's data. Denormalization of the database of the discussed case is justified if there are

many read requests on the tables Employee and Department, and these tables are updated rarely. Special databases – OLAP databases or data warehouses – for which the performance of read queries from multiple very large tables is important, are designed as denormalized. Denormalization of OLTP databases is considered if required performance cannot be achieved by other means, some of which are discussed below.

The performance of some queries can be improved by denormalization of the involved tables.

Another possibility for improving performance is to consider splitting a "wide" table into several "narrower" tables. In Chapter 2 it is explained that data from the table is read by data blocks. The "narrower" the row of a table, the more rows fit in the block. For accessing N rows of a narrower table the system has to read fewer blocks than for a wider table. However, this solution makes retrieving and modifying of the cross-table data more complicated and expensive, which is why it is used in special situations only. This approach can be applied when a table has one or more very long columns. For example, users need to keep the resumes of employees in a database. Adding the column resume to the Employee table can have negative impact on the performance of read queries – because of this long column each block of the table will contain only a few rows, and the number of blocks, which are accessed during query processing, will be significant. If instead the new table Employee_Resume (ID, resume) is created, then performance of existing queries will not be compromised. For this example, the data about resumes will be requested separately from the other data about employees and there will not be many cross-table queries. It is recommended to keep long columns in a separate table, especially if they are accessed separately from other columns of the table.

Consider keeping very long columns in a separate table.

It is important to choose appropriate data types for columns, especially the key columns. The key columns must be simple and short – numeric data types are the best choice.

Choose correct data types for columns. Make the key columns simple and short.

A distributed design of the database (see Chapter 3) can significantly improve the performance of some applications by localizing the data.

There is no unconditionally perfect database design. The design of the database is defined by its purpose, the types and frequencies of database requests, and performance requirements.

Design of Applications and Queries

Often performance problems can be caused by an inappropriately designed application or by a lack of understanding of the database features by application developers. For example, an application may be programmed without considering how the DBMS handles concurrent data access and may request exclusive access to data where it actually does not need it. As a result, this application will block the processing of other applications that need the same data resource or make its concurrent execution by several users impossible (concurrent access to data and its problems are explained in Chapter 6).

In another case, an application can make multiple unnecessary accesses to the database. Reducing the number of accesses can reduce the load on the database and make the performance of this and other applications better. For example, for the Manufacturing Company case users frequently have to access data about employees of particular departments. The application shows users the list of the company's departments,

allows them to select a department, and returns the data about employees of the selected department. One way to implement this application is to read the data about the departments for every such request. However, because the data about departments does not change often, a better way is to the read data about the departments once (e.g. when the application is started) and keep the data in memory (if needed, the data about the departments can be refreshed). This gives two advantages: firstly, the application will access data about the departments much faster, and secondly, there will be fewer trips to the database and less resource contention, which may improve the performance of other applications as well.

Here is another example of optimizing the performance of an application. Imagine that the application has to produce a report that requires multiple accesses to several tables retrieves and processes large amounts of data. These multiple data requests can be implemented in the application. As a result, the application performs multiple trips to the database, and data is transferred from the database to the application multiple times contributing to the network traffic. If, on the other hand, the report is implemented as a stored procedure and the application has the procedure call, then the application performs only one database access and receives only the final set of data (see Figure 3). This solution benefits not only the application that produces the report, but also other applications working on the database.

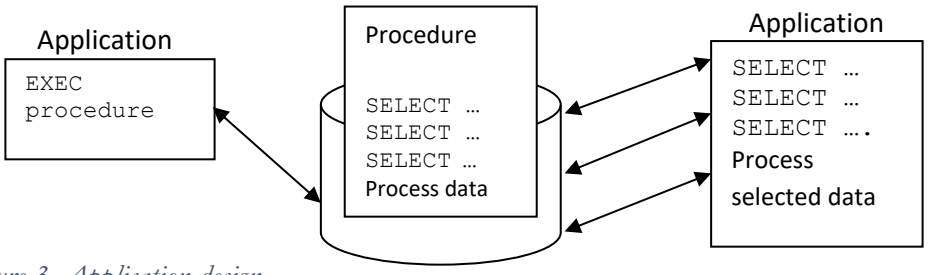

Figure 3. Application design.

Minimize the number of database accesses and amounts of data transferred from the database.

Queries on the database must be programmed in such a way that they benefit from the query processing features of a particular DBMS and take into consideration the design and implementation of the database. Though query optimizers today are significantly more "intelligent" than before and are less dependent on the query form to produce a good execution strategy, the role of the database programmer who understands how queries are processed and knows how the database is implemented, is very important. Here is an example of the importance of understanding of the query processing features of the DBMS. The query selects data about employees whose names start with the letter 'A' or 'a'. Assume that there is an index on the column emplName of the table Employee. Both of the following queries return the correct data. However, the first query is not able to use the index on the column emplName because of the applied function UPPER[2]. If the table Employee is large and it is necessary to use indexed access, then the first query is not a good solution:

```
SELECT * FROM Employee
```

[2] `UPPER` function in Oracle returns the string value of the argument converted to the upper case, e.g. `UPPER('abc') = 'ABC'`.

```
WHERE UPPER(emplName) LIKE 'A%';

SELECT * FROM Employee
WHERE emplName LIKE 'A%' OR emplName LIKE 'a%';
```

Another example shows that the programming of queries requires knowledge of how the database is designed and implemented. Any of the following queries can be used for reading the data about employees of the departments '001', '002', and '003':

```
SELECT * FROM Employee
WHERE deptCode BETWEEN '001' AND '003';

SELECT * FROM Employee
WHERE deptCode IN ('001', '002', '003');

SELECT * FROM Employee
WHERE deptCode = '001' OR deptCode = '002' OR deptCode = '003';
```

The query optimizer most probably will convert the second query into the third. Therefore, let us compare the first and third queries for different implementations of the table Employee:

- The table is stored as a heap. Whether there is an index on the column deptCode or not, the queries are processed similarly.
- The table is stored in the index cluster with deptCode as the cluster key. The queries are processed similarly.
- The table is stored in the hash cluster with the hash function applied to the column deptCode. The third query benefits from the cluster storage and accesses only blocks with data about the requested departments. However, the first query cannot take advantage of the cluster and performs full table scan. This happens because the expression deptCode BETWEEN '001' AND '003' of the first query is transformed into deptCode >= '001' AND deptCode <= '003', and hash clusters only can improve the performance of an exact equality queries. Therefore, for this case, the third query is the appropriate way to program the request.

These examples remind us about the effect of database transparency – we cannot rely on the application developers to know the details of the database implementation and the specifics of database processing. Therefore, it is responsibility of the database developer to provide the views or procedures that implement the application's requests in the most efficient way.

The documentation of each DBMS contains specific recommendations for writing efficient queries.

Data Storage

Physically data are stored in tables. The design and the implementation of tables requires attention and consideration because they have a major impact on performance. One of the goals of organizing data storage is to minimize the number of database blocks that are accessed during query processing (see the detailed discussion of data storage in Chapter 2). Several approaches help reduce the number of block accesses.

Effective Packing of Data in Tables

Data blocks must be as full as possible to store the maximum number of rows, however, there must be enough space left for updating the rows, which are already in the block. Depending on the nature of the data and database activities, database programmers and administrators have to find a compromise between

partially full blocks and the risk of having migrating rows.

Keep data effectively packed and avoid row migration.

Appropriately chosen data types can improve data packing, e.g. choosing the VARCHAR2 data type for columns with variable length can result in more rows per block than when using the CHAR data type.

Clustering

The performance of queries using equality or range conditions on a column (or columns) can be improved by clustering, where rows with the same value of the column are stored in the same data blocks. This type of clustering is called *index clustering*. The index cluster needs an index to support it and is accessed through this index. For example, if the table Employee is clustered by the deptCode column, then the processing of queries with the condition …WHERE deptCode = X will result in fewer block accesses than for the heap storage of this table where rows of the department X are scattered across numerous blocks. The performance of join queries can benefit from clustering the joined tables – this is like pre-joining rows of these tables and storing them together in a cluster. Consider the query:

```
SELECT emplName, deptName
FROM Employee e, Department d
WHERE e.deptCode = d.deptCode;
```

Its performance can be improved by storing the tables Employee and Department in a cluster with the cluster key deptCode.

Note that the performance of queries on one of the clustered-together tables may become worse because records of the table will be stored across more blocks than if the table were stored by itself. For example, the performance of the following query will be worse if the table is stored in a cluster together with the table Employee:

```
SELECT * FROM Department;
```

Most DBMSs also support another type of cluster – the *hash cluster*. Rows in the hash cluster are stored not based on the value of the cluster key, but based on the result of applying the hash function to the cluster key column. The hash function applied to the cluster key defines the location of a new row when data are inserted, and the location of an existing row when data are retrieved. Appropriately defined hash clusters allow accessing the row in a single disk read.

Hash clusters are beneficial for queries that have exact equality conditions. Range or non-equality queries, like. … WHERE deptCode <> '002' or … WHERE deptCode > '002' cannot benefit from hash clusters.

Though clusters can significantly improve performance and increase the database throughput for read queries, they can cause performance problems for tables that have many updates and inserts because of the necessity to reorganize clusters.

Consider clustering tables for improving the performance of queries with equality conditions and joins if the tables are not modified often.

Indices

With the heap organization of storage, the retrieval of data in many cases requires access to most of the blocks with data (this is called full table scan). For example, to find the employee with ID = 1 (ID is the

primary key and therefore unique), the system has to perform reading blocks of the table one after another until the requested row is found. The index cluster organization itself does not help much in improving performance of query processing if the system does not know in which blocks of the cluster the requested data are located. Indices are special database objects that make access to data more efficient (indices are explained in Chapter 2).

Let us perform a rough estimation of the second strategy for execution of the query about employees of the IT departments, but this time considering an index on the deptType column of the table Department and an index on the deptCode column of the table Employee. For simplicity, assume that the system performs access to one data block in the index in order to read the address of a row of the table. Remember that indexed access includes reading index blocks, finding the addresses of the needed table blocks, and then reading the respective table blocks. For the indexed execution of the discussed query, considering the assumptions:

1. For the selection, read the index on the deptType column of the table Department: 40 accesses, then read the rows of the table Department: 40 accesses.
2. For the intermediate result of the selection, write all rows to disk: 40 accesses.
3. For the join, read the result of the previous step: 40 accesses, then read the index on the deptCode column of the table Employee: 800 accesses, and then read the data of the table Employee: 800 accesses.

The total cost is significantly less than the costs of access without indices:

$$40 * 3 + 40 + 800 + 800 = 1760.$$

It is important to understand how indexed access to data is performed and that indices are not always beneficial. Let us compare the indexed and full table scan performances of the query, which requests data about employees that do not work in the IT departments:

```
SELECT ID, name
FROM Employee e, Department d
WHERE e.deptCode = d.deptCode AND deptType <> 'IT';
```

This query differs from the previously discussed query only by the condition on the deptType column. Estimation is provided for the following strategy:

$$\text{Employee} \blacktriangleright\blacktriangleleft_{\text{Employee.deptCode = Department.deptCode}} \sigma_{\text{Department.deptType<>'IT'}}(\text{Department}).$$

If there are no indices, the system performs the full scan of the tables:

1. For the selection, read the table Department:200 accesses. The selection results are 160 rows.
2. Write the intermediate result of the selection to disk: 160 accesses.
3. For the join, read the rows of the table Employee: 4000 accesses, and rows of the result of the previous step: 160 accesses.

The total cost of processing without indices is:

$$200 + 160 + 4000 + 160 = 4520.$$

The cost of index access is calculated as follows:

1. For the selection, read the index on the column deptType of the table Department: 160 accesses, then read the respective blocks of the table itself: 160 accesses.

2. Write the intermediate result of the selection to disk: 160 accesses.
3. For the join, read the rows of the result of the previous step: 160 accesses, then read the index on the column deptCode of the table Employee: 3200 accesses, and then access the respective rows of the table Employee: 3200 accesses.

The total cost of processing with indices is:

$$160 * 3 + 160 + 3200 + 3200 = 7040.$$

The total cost of the indexed access is significantly higher than the cost of access without indices. This example demonstrates that the index is not efficient when a large number of rows of the table are accessed. Indices are considered beneficial for accessing not more than 15- 25% of the rows of a table.

If indices can enhance the performance of read queries, they can decrease the performance of modifying queries, because in many cases not only does the table have to be updated, but also the indices of the table as well.

An index can be built on multiple columns, e.g. for frequent requests of the names of full-time employees working in a particular department, the following index may be useful:

```
CREATE INDEX i1 ON Employee(empType, deptCode);
```

Because the data in this index is organized by empType and within empType by deptCode (similarly to the last and first names in the phone directory), the index can be used for queries with conditions on both columns of the index or with conditions on the empType column only:

```
… WHERE empType = 'Full-time' AND deptCode = '001';
```

or

```
… WHERE empType = 'Full-time';
```

In most DBMSs this index cannot be used for queries with the condition on deptCode only:

```
… WHERE deptCode = '001';
```

If there are two groups of queries: queries of the first group are constrained by both empType and deptCode, and queries of the second group are constrained by deptCode only, then the second group of queries cannot use the previously created index i1. However, both groups of queries can benefit from the index:

```
CREATE INDEX i2 ON Employee(deptCode, empType);
```

Because indices occupy significant space[3] (some indices can be larger than their table), and because each index adds to the burden of processing of modifying queries, the number and types of indices must be carefully considered. For example, it does not make sense to keep both indices i1 and i2. The index i2 helps in the processing of the two groups of most frequent queries, and therefore we do not need to keep index i1 (note that since index i1 cannot improve the performance of all the mentioned queries, it should not be chosen in preference to i2).

[3] On average, the size of the indices is about 80% of the size of the tables.

The composite index is used for the processing of queries in which the leftmost part of the index columns is constrained.

Database programmers must take advantage of another important feature of indices: if a query requests only columns that are contained in the index, then there is no need to access the table at all, and the performance of such a query will especially benefit from the index. For example, the system uses only the index i2 to process the following query about the number of employees of each type in several departments:

```
SELECT empType, COUNT(*)
FROM Employee
WHERE deptCode IN ('001', '002')
GROUP BY empType;
```

The index can be sufficient for data access of queries that reference only the columns of the index.

In some cases, in order to avoid accessing the table, it is useful to create a multi-column index and have all the needed data in it. For example, though the index i2 is unnecessarily complicated for queries that access data from the table Employee given a department code, it is sufficient for any query that involves the attributes deptCode and deptType and, therefore, can be beneficial to the query's performance.

In some cases, the optimizer can take advantage of several different table indices. Assume, for example, that there are two indices on the table Employee:

```
CREATE INDEX i1 ON Employee(deptCode);

CREATE INDEX i2 ON Employee(emplType);
```

A query about employees of a particular department and a particular type can use both indices. With the help of the first index, the optimizer finds the rows of the table that have the requested department code, with the help of the second index – the rows of the specified employee type, and then finds the intersection of the two found sets of rows (processing of the retrieved rows takes place in memory).

You have to remember that the DBMS always builds an index on the columns of the primary key of a table; it is called the primary index. Indices on non-key columns are called secondary indices. When considering a secondary index, the primary index has to be taken into account. For example, if the table T (A, B, C, D) has the primary index on attributes A and B (in that order), we can have a number of situations for different query patterns based on attributes:

- A and C. Probably, instead of creating the index on the attributes A and C, a single-column index on the attribute C will suffice.
- B and C, and B and D. If queries constrained by the attribute A are not common, then changing the order of attributes in the primary index can help in building more efficient secondary indices.

It is recommended to consider a secondary index in the following situations:

- There are frequent read queries that involve constraints on certain table columns and the performance of these queries is important. Then the constrained columns could be used as the secondary index.
- Columns of the index are selective enough – queries constrained by these columns return not more than 25% of rows of the table.

- Updating activities on the table are not too heavy and their performance is less crucial than the performance of data retrieval.
- On the foreign key attribute of a child table when the child and parent tables are used together often in join queries.

Analyze the most frequent queries on a table to define the best combinations and order of columns in table's indices.

Avoid building indices for small tables and on long character columns.

Storage parameters of indices must be defined similarly to the storage parameters of tables. For better performance it is recommended to store the tables and their indices on different disks, so that the system can access the table and the index in parallel.

Index statistics must be computed on a regular basis for DBMSs that apply cost-based optimization use statistics on indices.

Other Factors

Taking care of the previously discussed application related factors is usually the responsibility of the developers of the database and database applications. There are other factors of a more global character, which influence the performance of the whole database and are important for multiple applications on the database. These are factors defined by the database environment, such as the storage subsystem, which includes multiple storage devices (disks or disk arrays), available memory, processing units, and the operating system.

Because of the global nature of these factors, planning, configuring and managing them is the responsibility of the system and database administrators. Tuning of the database environment is beyond the subject of this chapter, but for a better understanding of the organization of databases and query processing mechanisms, we need to discuss such parts of the database as the database buffers and log files.

Database Buffer

The database buffer and its role in the recovery of a database are discussed in more detail in Chapter 7.

Disk access is a very expensive operation in terms of performance, and the system tries to minimize the number of disk accesses. The ultimate solution to this would be to keep the whole database in memory and perform all the data processing there. Because it is not feasible for most applications, databases keep only some portion of their data in memory. The part of the memory dedicated to keeping data is called the database buffer. The size of the buffer is limited by the size of memory.

The logic of the buffer read typically is as follows:

- Go to buffer cache and look for the block.
- If the block is not there, perform physical I/O and put it into the cache.
- Return the block from cache.
- If the next user needs this block, he probably will find it in the cache.

However, there is the opportunity to do "direct I/O", normally used with parallel query - your server process will bypass the buffer cache and read right from disk into its own memory and process the blocks there.

Generally used for a large table full table scan.

Because it is impossible to fit the whole database in memory, the question is, which data should be kept in the buffer? Usually the database keeps the most recently accessed data in memory. Earlier in this chapter, for estimating different strategies, it was assumed that intermediate results of the relational operations were stored on disk. In reality, intermediate results usually are kept in the database buffer. The more that data are accessed from the buffer (and not from disk), the better the performance is. Data access from the buffer is called logical access (LA), and access from disk is called physical access (PA).

When the database is started, the buffer is empty (it is said that the database is cold) and most data accesses are physical. The performance of a cold database is worse than the performance of a warm database where the database has been used for some time and part of the data is in the buffer.

The database administrator has to define the size of the buffer based on the needs of all the applications on the database and the physical parameters of the server. A larger buffer can accommodate more data, which can bring a significant improvement of performance.

Separate Storage of Tables, Indices, and Log Files

Locating tables and indices on different disks allows the system to parallelize access to the index and access to the table when leveraging indices in query processing.

Log files are discussed in more detail in Chapter 7. The purpose of log files is to protect data in the database buffer that has been modified recently from being lost, e.g. in the case of a system crash. All data modifications are recorded in log files. Writing to the log file can create a significant overhead, and hence it is important to configure log files appropriately, e.g. performance can be improved, if log files and tables are located on different disks (note that locating the log files on the same disks as tables defies the purpose of the log files to enable the data recovery – they will be lost together with the tables in case of the media crash). In some situations in order to improve performance, logging can be turned off.

Out of the Box Approaches

Database performance is affected by various organizational measures and activities. In some cases a simple, out-of-the-box approach helps to achieve the required performance.

For example, many times during a business day an accounting application requests yesterday's account balances. The computation of the balances is complicated and resource consuming, and tuning efforts do not give the required performance. In this situation, it may be a good idea to compute balances every day and store them in a separate table for the next day activities.

Another example shows how performance problems can be resolved by organizational measures. Pre-computing account balances for the previous example is resource consuming and creates resource contention; and hence it may be reasonable to execute it after the end of the business day when the load on the database is low. The procedure will not interfere with other important applications that have to be executed during the day.

Examples of Influence of Different Factors on Performance

The following examples are taken from [Shasha] they provide quantitative demonstrations of how different factors influence database performance.

Figure 4 shows how the performance of a multipoint query[4] is dramatically improved by clustering: the throughput for the clustered table is about two times higher (depending on the DBMS used) than the throughput for a heap table with an index. For the 1 million-row table, which was used for testing, the performance on the heap table without the index is extremely poor.

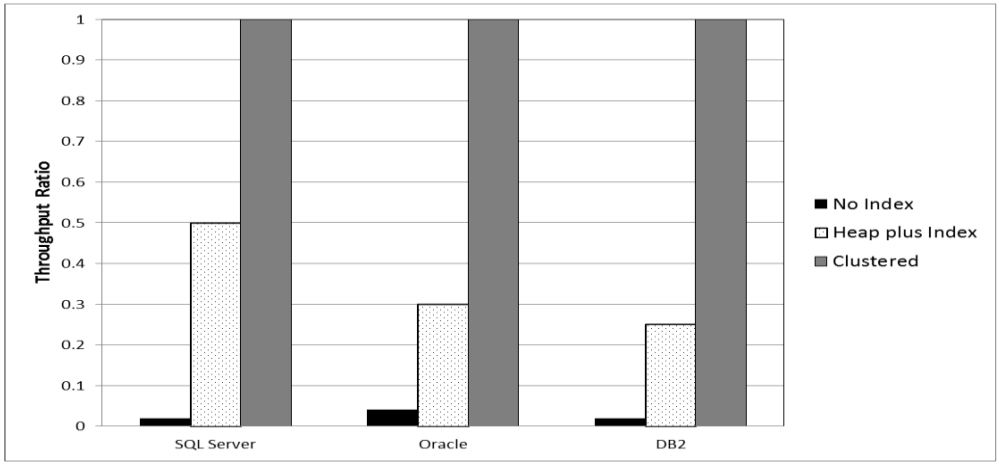

Figure 4. Clusters and Indexes.

Figure 5 shows how the index can worsen the performance of insert queries. The test involved 100,000 insertions using Oracle for a table with and without an index.

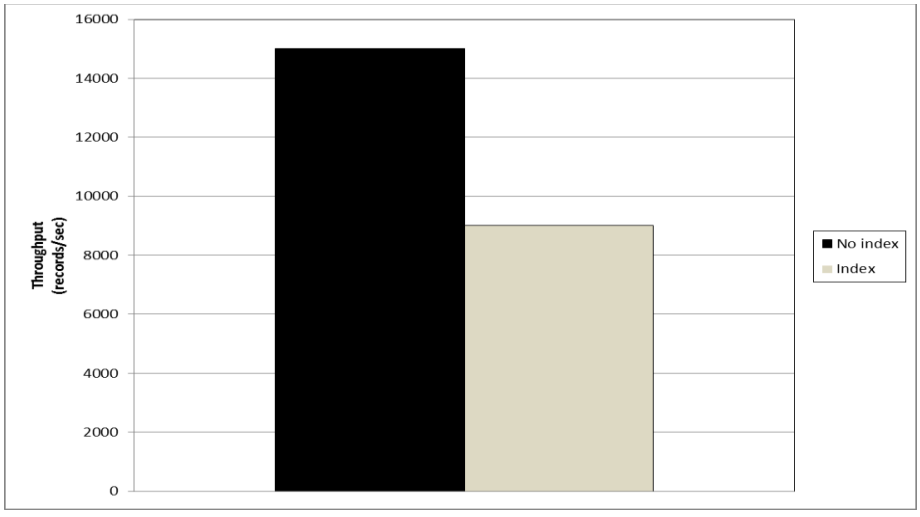

Figure 5. Index overhead for insertions.

[4] A multipoint query is a query that returns multiple records for an equality condition.

Figure 6 demonstrates the performance improvement with an increase in the buffer size. The performance of the multipoint query increases linearly with the increasing buffer size until the whole table fits in memory.

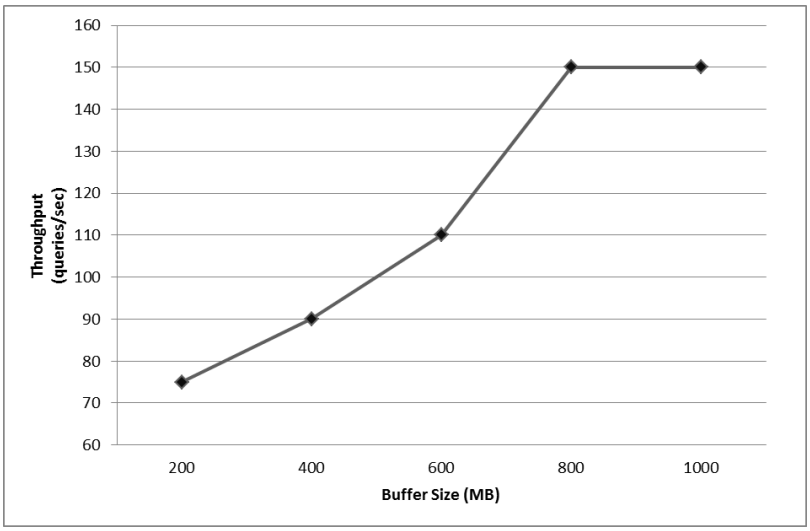

Figure 6. Buffer size.

Figure 7 shows that locating the log file and the table on different disks improves the performance of insertions and updates by almost 30%.

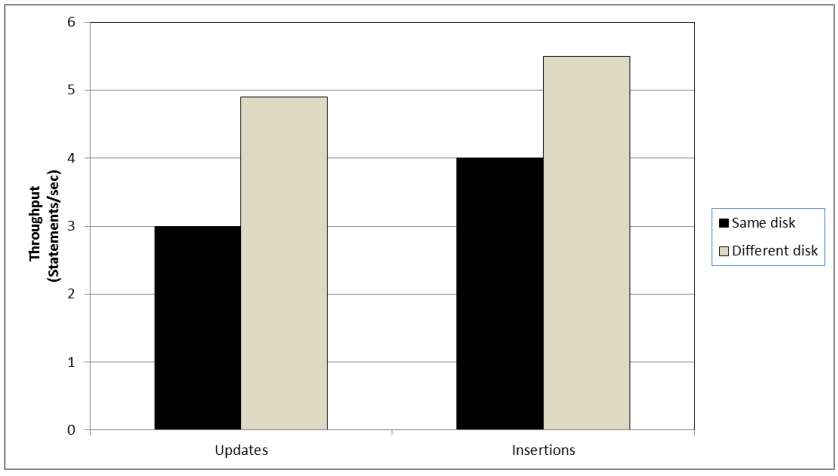

Figure 7. Location of the log file.

Database Tuning

Activities directed towards the improvement of database performance are called database tuning. Tuning of the database starts at the design stage and continues to the maintenance stage.

Proactive Tuning

Design, development and implementation efforts are directed towards achieving the performance required by users. This is proactive tuning of the database, because designers and developers, as well as administrators, try to meet users' expectations. Proactive tuning defines to a large extent the cost of further database maintenance. Figure 8 shows that decisions made at the earlier stages of database design and implementation have a significant impact on the performance of the database, while the cost at these stages is relatively low. Efforts directed towards the improvement of performance at the later stages have a higher cost and lower impact on database performance.

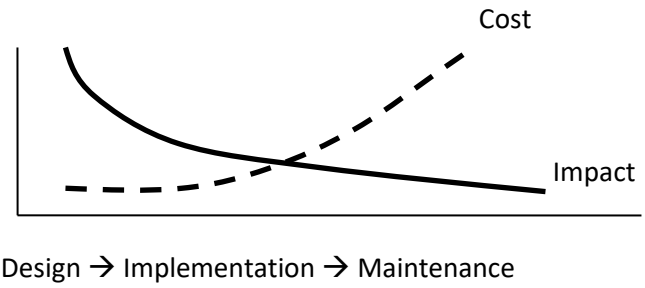

Figure 8. Impact and cost of tuning at different stages of database life cycle.

The recommended sequence of proactive tuning steps is the following:

- *User requirements*. Sometimes user expectations are unrealistic or unnecessarily high. Relaxing user requirements without sacrificing the functionality of the database can reduce the cost of the database design, implementation and maintenance.
- *Data design*. The design of the data must correspond to the nature of the database applications.
- *Application design*. Applications should not create an unnecessary load on the database; the number of database accesses and the amount of transferred data should be kept to a minimum.
- *Queries*. Queries must be programmed such that they take full advantage of the SQL features of the DBMS and the query optimizer, and must be developed with an understanding of the database design and implementation.
- *Access to data*. The number of physical data accesses must be minimized. All necessary indices must be created, clustering should be considered, the allocation of data in data blocks must be optimized, and the size of the database buffer must be increased.
- *Database environment*. The best configuration of memory, storage and processing units must be defined.

Monitoring and Troubleshooting

During the lifetime of a database, some of the database features or settings of the environment that were predefined at the development and implementation stages may become inconsistent with the current state of the database or the way the database is used. For example, a table may become very large and the performance of some queries may decrease which might require reorganizing the table storage. Or, the number of concurrent users of the database may increase and lead to lower throughput; in either case upgrading the database environment and rewriting the database transactions may be necessary.

Monitoring performance of the database is the responsibility of the database administrator. Most DBMSs

provide powerful tools for performance monitoring and controlling the consumption of resources. Analyzing the results of monitoring enables the administrator to diagnose problems and bottlenecks, and to define ways of dealing with these issues. Fixing the problems may require the intervention of the database designer or developer.

Tuning the database is often a series of compromises, e.g. you may sacrifice the performance of less important queries to improve the performance of the crucial ones. Or, if the performance analysis indicates that we need an expansion of disk storage or memory and it is not possible due to technical, financial or political reasons, we may need to limit the number of concurrent users.

Performance in Distributed Databases

Optimization of Distributed Queries

Improving performance is the primary goal of data distribution. The appropriate distribution should significantly improve the performance of local applications and queries. Further improvements in the performance of the local databases are provided as in any centralized database.

Improving the performance of global queries, however, may require special measures. Let us discuss how query processing is performed in the distributed database and the possible ways to improve it. Making the distributed database transparent to users was discussed in Chapter 2. In the transparent distributed database, global users define their requests on global external views. To execute a query, the system has to transform the query on the views into the query on the fragments and unfragmented relations distributed across the various databases that make up the distributed database.

Let us assume that the database for the Manufacturing Company case is distributed as in the example of Chapter 3: the table Department is fragmented by the attribute location, and the fragmentation of the table Employee is derived from the fragmentation of the table Department. The global applications are executed from the New York site. The query about the ID and name of employees of the IT departments that was discussed in this chapter for the centralized database is formulated on global views for the distributed database:

```
SELECT ID, name
FROM Employee e, Department d
WHERE e.deptCode = d.deptCode AND d.deptType = 'IT';
```

Consider the global strategy similar to the second strategy for the same query in the centralized database:

$$\text{Employee} \blacktriangleright\blacktriangleleft_{\text{Employee.deptCode = Department.deptCode}} (\sigma_{\text{Department.deptType='IT'}}(\text{Department})$$

This query must be transformed into a query on fragments and relations in different locations using the fragmentation reconstruction rules:

$$(\text{Employee}_1 \cup \text{Employee}_2 \cup \text{Employee}_3) \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}}(\text{Department}_1 \cup \text{Department}_2 \cup \text{Department}_3))$$

Since the global query is run in New York, data are requested from the New York database, and there are several ways to execute the query. For example:

- Send all necessary data to the New York database and then perform all processing there following the expression above.
- Distribute the query, perform local processing on each involved site, then send the results to the New York database and finish the processing there. For this approach, the strategy may be rewritten as (the detailed explanation of how the following expression is obtained is given in earlier):

$$\text{Employee}_1 \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}}(\text{Department}_1)) \cup$$

$$\text{Employee}_2 \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}} (\text{Department}_2)) \cup$$

$$\text{Employee}_3 \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}} (\text{Department}_3)))$$

This strategy performs local selections on the Department fragments and joins the results to the local fragments of Employee:

$$\text{Local result}_i = \text{Employee}_i \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}}(\text{Department}_i)) \text{ [for i=1,2,3]}$$

Then the results of the local processing are transferred to the New York database; the New York database performs the union of the local results:

$$\text{Local result}_1 \cup \text{Local result}_2 \cup \text{Local result}_3$$

The cost of processing of distributed queries includes the cost of data transfer between the databases.

The total Cost =
(the number of I/O operations * the cost of one I/O operation) +
(the number of RAM operations * the cost of one RAM operation) +
(the number of transferred bytes * the byte transfer cost)

The data transfer cost is usually considerably higher than the cost of I/O operations, therefore the following rough estimation of the strategies does not consider the cost of memory and I/O operations. The cost of data transfer is calculated as the number of transferred rows.

Assume that the data are distributed evenly across the sites and on each site there are approximately 65 departments and 1330 employees. The first approach results in approximately 2660 row transfers for data of employees and 130 row transfers for the departments from both the Boston and Cleveland sites. The total cost is 2790.

For the second approach, each site performs the local selection of the IT departments and joins the result of the selection with the data about the local employees. For each site, there are 13 IT departments with 20 employees in each of them. The total cost of data transfer from the Boston and Cleveland sites is 520. Obviously, this approach gives better performance. Besides, in this case smaller amounts of data are transferred across the network – this reduces resource contention and improves the performance of other applications.

Distributed query processing includes two additional steps (Figure 9):

- *Data localization*. The query on global views is transformed into a query on fragments and unfragmented relations (or replicas).

- *Global optimization*. The system develops a global strategy for distributed query processing. The strategy is directed towards the minimization of the data transfer cost.

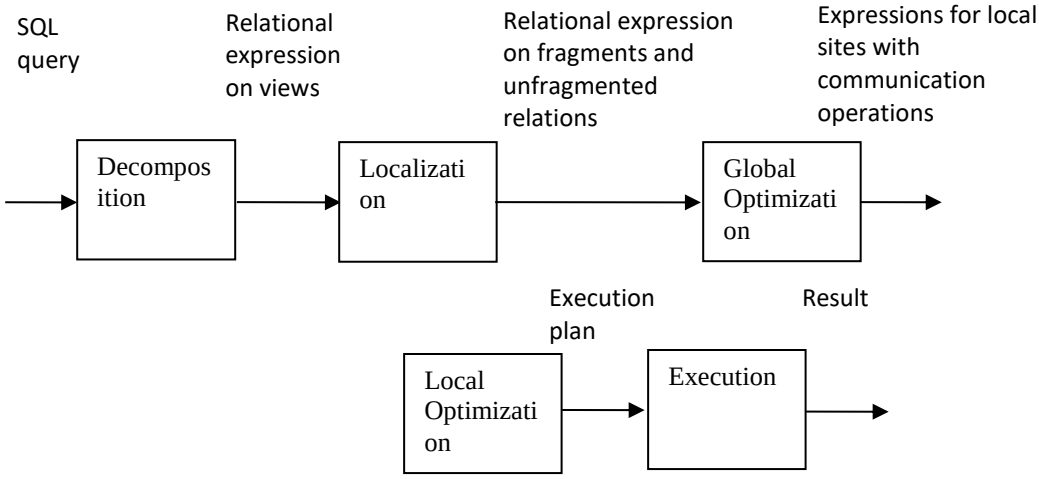

Figure 9. Scheme of distributed query processing.

Reduction

The localization of the distributed query in the previous section was performed by application of the reconstruction rules: the query on global views was transformed into a query on fragments. The resulting query is called the *generic* query.

For some distributed requests, generic queries can be simplified or reduced – this results in creation of new, reduced views. The following sections provides some examples of reduction.

Reduction of the Primary Horizontal Fragmentation

Let us consider the query, which requests data about the New York and Cleveland departments:

```
SELECT * FROM Department
WHERE location IN ('Cleveland', 'New York');
```

The query is decomposed into the following relational expression on global views:

$$\sigma_{\text{location IN ('Cleveland', 'New York')}}(\text{Department}).$$

The view Department is implemented as the union of the horizontal fragments:

$$\sigma_{\text{location IN ('Cleveland', 'New York')}}(\text{Department}_1 \cup \text{Department}_2 \cup \text{Department}_3).$$

One of the transformation rules for the relational operations says that the selection on the union of relations is equal to the union of selections on relations, therefore the generic query can be rewritten as:

$$\sigma_{\text{location IN ('Cleveland', 'New York')}}(\text{Department}_1) \cup$$

$\sigma_{\text{location IN ('Cleveland', 'New York')}}(\text{Department}_2) \cup$

$\sigma_{\text{location IN ('Cleveland', 'New York')}}(\text{Department}_3)$.

Because the condition of the selection from the fragment Department_1 (location = 'Boston') is conflicting with the predicate of the fragment, the result of applying the selection on the fragment Department_1 is always empty. Therefore, the generic query can be reduced to:

$\sigma_{\text{location IN ('Cleveland', 'New York')}}(\text{Department}_2) \cup$

$\sigma_{\text{location IN ('Cleveland', 'New York')}}(\text{Department}_3)$.

Reduction of the Derived Horizontal Fragmentation

In some cases, the processing of distributed queries that involve both primary and derived fragments can be significantly simplified. Consider the query about the ID and name of employees from the IT departments from earlier. The generic query for this query is:

$(\text{Employee}_1 \cup \text{Employee}_2 \cup \text{Employee}_3) \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}}(\text{Department}_1 \cup \text{Department}_2 \cup \text{Department}_3))$

The transformation rule for the join of unions says:

$$(R_1 \cup R_2) \blacktriangleright\blacktriangleleft (S_1 \cup S_2) = R_1 \blacktriangleright\blacktriangleleft S_1 \cup R_1 \blacktriangleright\blacktriangleleft S_2 \cup R_2 \blacktriangleright\blacktriangleleft S_1 \cup R_2 \blacktriangleright\blacktriangleleft S_2$$

With the help of this rule, the generic query is transformed into:

$\text{Employee}_1 \blacktriangleright\blacktriangleleft \text{Department}_1 \cup \text{Employee}_1 \blacktriangleright\blacktriangleleft \text{Department}_2 \cup$

$\text{Employee}_1 \blacktriangleright\blacktriangleleft \text{Department}_3 \cup \text{Employee}_2 \blacktriangleright\blacktriangleleft \text{Department}_1 \cup$

$\text{Employee}_2 \blacktriangleright\blacktriangleleft \text{Department}_2 \cup \text{Employee}_2 \blacktriangleright\blacktriangleleft \text{Department}_3 \cup$

$\text{Employee}_3 \blacktriangleright\blacktriangleleft \text{Department}_1 \cup \text{Employee}_3 \blacktriangleright\blacktriangleleft \text{Department}_2 \cup$

$\text{Employee}_3 \blacktriangleright\blacktriangleleft \text{Department}_3$.

Because of the nature of the derived fragmentation, any join $\text{Employee}_i \blacktriangleright\blacktriangleleft \text{Department}_j$, where $i \neq j$ produces an empty result. Therefore, the above expression can be reduced to the expression that was mentioned before:

$\text{Employee}_1 \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}}(\text{Department}_1)) \cup$

$\text{Employee}_2 \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}}(\text{Department}_2)) \cup$

$\text{Employee}_3 \blacktriangleright\blacktriangleleft_{\text{deptCode}} (\sigma_{\text{deptType='IT'}}(\text{Department}_3))$.

Reduction of the Vertical Fragmentation

Let us discuss the vertical fragmentation of the relation Employee from the Case 3.6 of Chapter 3. Let's say that the New York users need data about the names of all employees and the codes of the departments to

which employees are assigned. In addition, let's assume that the Cleveland users are working with data about the types and title codes of all employees.

$$Employee_1 = \Pi_{ID,\ empName,\ deptCode}(Employee)$$

$$Employee_2 = \Pi_{ID,\ empType,\ titleCode}(Employee).$$

The query requests all titles that are used by employees of the company:

```
SELECT DISTINCT titleCode
FROM Employee;
```

This query is decomposed into:

$$\Pi_{titleCode}(Employee).$$

Reconstruction of the global relation from vertical fragments produces the following generic query:

$$\Pi_{titleCode}(Employee_1 \blacktriangleright\blacktriangleleft_{ID} Employee_2).$$

One of the transformation rules says that the projection on the join of relations is equal to the join of projections on relations:

$$\Pi_{titleCode}(Employee_1 \blacktriangleright\blacktriangleleft_{ID} Employee_2) =$$

$$\Pi_{titleCode}(Employee_1) \blacktriangleright\blacktriangleleft \Pi_{titleCode}(Employee_2).$$

It is easy to see that the projection on the fragment $Employee_1$ produces the empty result and can be eliminated from the query. The resulting reduced query is:

$$\Pi_{titleCode}(Employee_2).$$

Reduction of the Hybrid Fragmentation

The reduction of the hybrid fragmentation is provided by the application of the discussed reduction approaches:

1. Remove the selection on a horizontal fragment if the selection condition conflicts with the predicate of the fragment.
2. Remove the projection on a vertical fragment if the fragment does not contain any of the columns of the projection.
3. Remove joins, which produce empty relations.

For distributed queries that allow reduction, it is useful to apply the reduction rules and build reduced global views, i.e. views that include only fragments or relations that are relevant for the query.

Query Processing in Oracle

Types of Optimization in Oracle

Oracle supports two types of optimization: the rule-based (RBO) and the cost-based (CBO). CBO was first

introduced in Oracle 8 and now is considered the main optimization type. However, CBO can be applied only if the necessary statistics are available.

Rule-Based Optimization

To produce a strategy of query execution, RBO applies heuristic rules. It uses information about the structure and integrity constraints of the tables involved in the query, existing indices and their types (primary, secondary, cluster, single-column, composite). Rules rank different types of access to data: the lower the rank, the better the performance is expected. The highest rank is for a full table scan (all rows of the table are accessed), the lowest – direct access to a row with the help of the physical address of the row (ROWID). The optimizer tries to apply the lowest-rank operations first. For example, if there is an index on the column deptType of the table Department and an index on the column deptCode of the table Employee, then for the query

```
SELECT *
FROM Employee e, Department d
WHERE e.deptCode = e.deptCode AND deptType = 'Business';
```

RBO decides to use the index on the column deptType of the table Department because the rank of the single-column index is lower than full table scan. Further, to access respective rows of the table Employee for selected department rows, the optimizer uses the index on the column deptCode.

Cost-Based Optimization

CBO is recommended for most applications and is used as the default when possible (the optimizer can apply it only if database statistics are available). Statistics can be calculated by the statement ANALYZE or with the help of the supplied package DBMS_STATS. For example, the following statements gather the statistics for the table Department and one of its indices:

```
ANALYZE TABLE Department COMPUTE STATISTICS;

ANALYZE INDEX i0 COMPUTE STATISTICS;
```

Statistics must be refreshed on a regular basis because obsolete statistics can cause the optimizer to not choose the best strategy for the query execution for the current state of the database.

In addition to the usual statistical data mentioned earlier, Oracle supports histograms. Histograms contain the most detailed information about the distribution of data in columns and they allow for a more accurate estimation of execution strategies than regular statistics. For example, suppose there are 10 different department types and that of the 200 departments, only 60 departments are of the business type. Regular statistics on the table Department say that for 200 rows of the table there are 10 distinct values of the column deptType. Based on this information, the optimizer estimates that there are approximately 20 rows for each department type, and when building the strategy of execution of the query about employees of the business departments from the previous section, decides to use the index on deptType assuming that approximately 10% of the rows of the table will be accessed.

The histogram on the column deptType of the table Department (Figure 10) shows that there are 60 rows of the table, for which the column deptType is equal 'Business'. The optimizer may choose not to use the index and perform a full table scan of Department because based on the histogram it makes a conclusion that more than 25% of the table rows will be accessed.

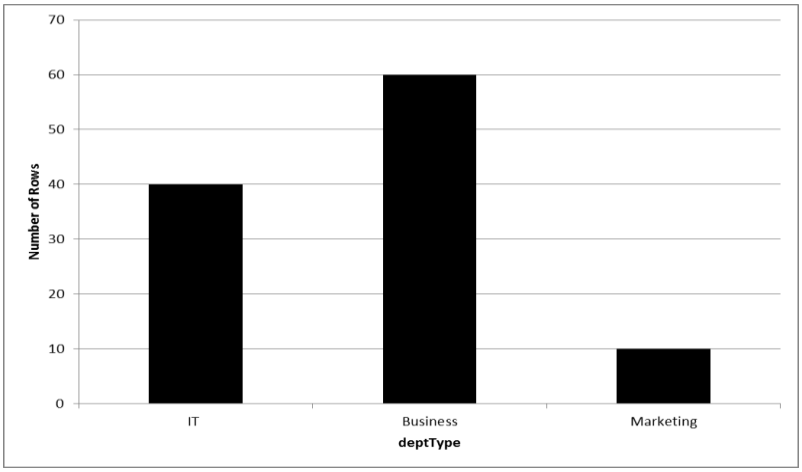

Figure 10. Histogram.

Information about statistics is available in the views of the database dictionary. Some of these views are:

- USER_TABLES contains the statistics of the tables owned by the user.
- USER_TAB_HISTOGRAMS contains the information about the histograms for the tables.
- USER_TAB_COL_STATISTICS contains statistics on the columns of the tables.

The following query returns basic statistics (the number of rows, the number of blocks, the average row length, and the date when the statistics were gathered) on the tables Department and Employee:

```
SELECT TABLE_NAME, NUM_ROWS, BLOCKS, AVG_ROW_LEN,
       TO_CHAR(LAST_ANALYZED, 'MM/DD/YYYY HH24:MI:SS')
FROM USER_TABLES
WHERE TABLE_NAME IN ('DEPARTMENT','EMPLOYEE');
```

TABLE_NAME	NUM_ROWS	BLOCKS	AVH_ROW_LEN	LAST_ANALYZED
DEPARTMENT	200	30	449	07/29/1999 00:59:51
EMPLOYEE	4000	50	663	07/29/1999 01:16:09

Elements of Dynamic Optimization

In earlier database releases, once the execution plan was determined it was followed during the runtime.

Adaptive Plans in Oracle Database 12c allow runtime changes to execution plans. In addition to the plan that is considered for execution (so-called default plan), the the optimizer also considers alternative subplans for each major join operation in the plan. At runtime the efficiency of the plan is checked using statistics and an alternative plan can be chosen to continue the execution.

Reusing Execution Strategies

Often many database queries have the same execution pattern. For example, in the Manufacturing Company case users often need to get data about an employee given the employee's ID. Obviously, the execution strategy for all queries of this type is the same. Optimization of each query adds to the query execution cost, therefore, performing optimization once and then reusing the produced strategy in subsequent queries of the same type can additionally improve performance.

The optimizer can reuse the execution strategy after an authorization check (that has to be performed for each query). Imagine that two users John and Scott try to execute the same query:

```
SELECT * FROM John.Table1;
```

If Scott does not have authorization to access John's Table 1, the authorization check will stop further processing; if Scott does have the authorization the optimizer will use the same execution strategy.

Oracle stores execution strategies of the most recent queries in a special part of memory, called the shared pool. When a new query is processed, Oracle performs syntax, semantic, and authorization checks of the query, and then checks the shared pool. If the same query was executed not long ago and its strategy is still stored in the shared pool, Oracle reuses the execution strategy and saves on optimization and generation of the execution strategy. Such processing is called soft parsing (as opposed to hard parsing when all the steps of query processing must be performed). Reducing the number of hard parses is very important for performance on a database with a large number of concurrent users issuing numerous requests of the same type. Utilizing this feature requires a special approach to queries programming.

The database cannot recognize that two requests have the same execution strategy if they are not identical strings. For example, a user is requesting data about the employee with ID = 1:

```
SELECT *
FROM Employee
WHERE ID = 1;
```

If the user requests data about an employee with a different ID later on, the optimizer will fail to match the query with the previously executed one, though the strategy of access definitely can be reused. Therefore, it is recommended to submit similar queries in the same form using the Oracle feature of bind variables. Instead of hard coding the value of an employee's ID in each query, the value is passed to the query through a variable; the variable is assigned the value before every query execution. The above query is rewritten with the help of a simple PL/SQL block with the bind variable:

```
VARIABLE v_id NUMBER;  -- Declaring the variable
BEGIN
   :v_ID := 1;         -- Assigning the value to the variable
END;

SELECT *               -- Using the variable in the query
FROM Employee
WHERE ID = :v_ID;

BEGIN
   :v_ID := 2;       -- Assigning a new value to the variable
END;
```

```
SELECT *                        -- Re-executing the query
FROM Employee
WHERE ID = :v_ID;
```

The request for data about another employee is formulated in exactly the same form as the first request, and the optimizer is able to apply the soft parse.

Most programming interfaces to Oracle support this feature, e.g. JDBC – Java Database Connectivity.

Data Storage

Different types of data storage and storage parameters are described in Chapter 2. The organization of data storage plays an important role in database tuning. Table storage is defined in the CREATE TABLE statement:

```
CREATE TABLE Department (
      deptCode CHAR(3) PRIMARY KEY,
      deptName VARCHAR2(20) NOT NULL,
      location VARCHAR2(25) CHECK (location = 'New York'),
      deptType VARCHAR2(15))
   PCTFREE 10
   PCTUSED 40
   TABLESPACE users
   STORAGE (  INITIAL 50K
              NEXT 50K
              MAXEXTENTS 10
              PCTINCREASE 25 );
```

The TABLESPACE clause specifies the tablespace where the table is located, and the STORAGE parameters specify the initial size and expansion of the table.

Parameters PCTFREE and PCTUSED specify how rows of the table are stored in data blocks. PCTFREE defines the percent of free space left in each block for possible future updates of rows of the block in order to avoid migrating or chained rows. PCTUSED sets the percent of used space in the block; a block with its percent of used space above the setting of PCTUSED is not considered available by the system for inserting new rows. The settings of the both parameters depend on the nature of the data in the table and the operations on the data, and they have an impact on the performance of read and write queries.

The value of PCTFREE cannot be set too high or the block space will be used inefficiently, and to prevent migrating rows it cannot be set too low. A too high value for PCTUSED will prevent unused space in the database but will result in more overhead for insert and delete operations because the block will be moved more frequently d from and to the list of blocks available for insertions. If the value for PCTUSED is too low, it will reduce the overhead of handling the list of available blocks, but will cause unnecessary expansion of the table.

For a table with a fixed row length and not many updates, the setting of PCTFREE has to be low; if a table has columns with variable length and their updates are possible, then the value of this parameter has to be higher. For a table with no or little deleting and inserting activities, the setting of PCTUSED can be high; for a table with many inserts and deletes, PCTUSED has to be lower. For example, for a table with a fixed row length and many expected delete and insert operations PCTFREE = 5 and PCTUSED = 50 are

recommended; for a large read-only table PCTFREE = 5 and PCTUSED = 90 are recommended; for a table with a variable row length, frequent updates and no delete operations PCTFREE = 30 and PCTUSED = 70 are recommended. The total of these two settings should not exceed 100.

Index storage parameters are defined similarly to storage parameters of tables.

Clusters

Oracle supports two types of clusters: index and hash. Clusters improve the performance of some read queries, but intensive updating activity can result in the necessity to reorganize clusters and will negatively affect the performance of updating queries. Oracle clustered storage is explained in Chapter 2.

In index clusters, rows of a table that have the same value for a column (which is called the cluster key) are stored in the same blocks. Such storage is beneficial for read queries with equality or range conditions on the cluster key attribute. For example, queries about employees of a particular type:

```
SELECT * FROM Employee WHERE emplType = X;
```

can benefit from a cluster with the column emplType as the cluster key because rows with the same employee type will be stored together, and the system can read all requested rows from one or several blocks. Clustered storage is prepared in accordance with storage parameters for the cluster that are similar to the storage parameters of tables:

```
CREATE CLUSTER Employee_type (emplType VARCHAR2(10))
TABLESPACE users
PCTUSED 80
PCTFREE 10
SIZE 40
```

The parameter SIZE specifies the length of the row and defines the number of rows in the block of the cluster. An unreasonably high value of the parameter can cause rows of the cluster to be stored across more blocks than necessary. A low value of the parameter can lead to chained data.

After the cluster is prepared, it is used for storing tables:

```
CREATE TABLE Employee (
      ID NUMBER PRIMARY KEY,
      . . .
      emplType VARCHAR2(10),
      . . . )
CLUSTER Employee_type (emplType);
```

Index clusters are supported by the index on the cluster key:

```
CREATE INDEX i0 ON CLUSTER Employee_type;
```

Index clusters are also useful for join queries when the parent and child tables are stored in the cluster with the foreign key as the cluster key.

Hash clusters are created similarly to index clusters, however, they do not need the cluster index. The hash function, applied to the cluster key, defines the address of the row. Hash clusters are beneficial for queries

with exact equality conditions only. To improve the performance of read queries that request data about employees of a particular type, the table Employee can be stored in a hash cluster with the hash function applied to the column emplType:

```
CREATE CLUSTER Employee_type_hash (emplType VARCHAR2(10))
TABLESPACE users
PCTUSED 80
PCTFREE 10
HASHKEYS 3;
```

Here are some recommendations about using clusters:

- Cluster together tables that often participate in join queries; in this case the foreign key column is chosen as the cluster key. However, clustering can make the performance on one of the clustered tables worse than when the table were stored separately.
- Index clusters can improve the performance of queries with range or equality conditions on the cluster key attribute.
- Hash clusters can be beneficial for queries with equality conditions on the cluster key attribute; they do not enhance performance of range queries.
- Clustering of tables with high modifying activity is not recommended.

Indices

The role of B-tree and hash cluster indices in improving performance was discussed before in this chapter. Oracle also supports a special – bitmap – type of index.

Bitmap Indices

Bitmap indices are beneficial for columns with low cardinality. They can dramatically improve query performance. Their additional advantage is that they are considerably smaller than B-tree indices.

The column emplType in the Employee table that has only three values is a good candidate for the bitmap index:

```
CREATE BITMAP INDEX i1 ON Employee(emplType);
```

The table below illustrates the bitmap index for the emplType column of the table Employee of our case. It consists of three separate bitmaps, one for each employee type.

emplType = 'Full-time'	**emplType = 'Part-time'**	**emplType = 'Consultant'**
1	0	0
0	0	1
0	1	0
1	0	0
0	0	1
1	0	0
0	1	0

Each entry or bit in the bitmap corresponds to a single row of the Employee table. The value of each bit

depends on the value of emplType in the corresponding row in the table. For instance, the bitmap **emplType = 'Full-time'** contains '1' as its first bit in the first entry of the index because the first row of the Employee table contains data about a full-time employee. The bitmap **emplType = 'Full-time'** has '1' in the fourth and the sixth rows of the index.

The bitmap index can efficiently process the following query about the number of full-time employees by counting the number of '1' in the **emplType = 'Full-time'** column of the bitmap. This query will not need to access the table at all:

```
SELECT COUNT(*)
FROM Employee
WHERE emplType = 'Full-time';
```

Because of specifics of their storage, bitmap indices can create performance problems for modifying queries. These indices are widely used in data warehouses (OLAP databases).

It is important to remember that indices enhance data retrieval; however, they can cause performance problems for modifying queries. Here are some general recommendations for using indices:

- Consider bitmap indices for columns with low cardinality.
- Create the index on the foreign key of a child table if the parent and the child tables are often used together in queries.
- Create the B-tree index for queries with equality or range conditions (e.g. salary < 30000) and prefix match queries (e.g. emplName LIKE 'A%').
- Avoid indexing long columns and creating indices on small tables.
- Do not create indices for queries that access more than 25% of rows in a table.

Function-Based Indices

Before Oracle 8, a function applied to an indexed column in a query disabled indexed access to data. For example, though we have the index on the column emplName of the table Employee, the following query will result in a full table scan because of the function UPPER:

```
SELECT * FROM Employee WHERE UPPER(emplName) LIKE 'A%';
```

This simple query can be rewritten to enable using the index, however, in situations that are more complicated, such limitation creates inconveniences. Now Oracle supports function-based indices that are created not on columns of a table, but on functions applied to columns. Such indices can be utilized by the optimizer to access the table's data for queries with conditions that include functions. The following function-based index can be used for improving the performance of the above query:

```
CREATE INDEX i0 ON Employee(UPPER(emplName));
```

Function-based indices can be based on both built-in and user-defined functions.

Partitions

Oracle partitions implement separate and physically independent storage for parts of a table. Such a storage arrangement is beneficial for queries that access rows from a particular part – the system's data search is more directed and efficient. For example, queries about employees of a particular type can benefit from partitioning the table Employee by emplType:

```
CREATE TABLE Employee (
       …
)
PARTITION BY HASH(emplType)

PARTITIONS 3;
```

This was an example of hash partitioning. Range partitioning can be applied to numeric and date columns. Partitioning by dates is very efficient in historical databases or data warehouses. For example, if a table stores data about daily activities of the business, then the analysis of the business activities for different years can benefit from partitioning the table's rows by year.

Storage parameters of each partition are specified similarly to storage parameters for tables. Local indices on a partition enforce a search on that partition. In addition to local indices, global table indices support a search on the whole table.

Index-Organized Tables

The index-organized table can be perceived as storing the primary index on a table with all the table's columns. Index-organized tables are extremely beneficial for queries with conditions on the key columns of the table – the system performs an indexed search and gets all the needed data in one read operation (compared to a table with an index, where the system reads the index and then reads the table). Another advantage of the index-organized table is that it occupies much less storage space than the table with its primary index.

Queries constrained by the non-key columns of the index-organized table do not benefit from such storage.

Tuning of Queries

Understanding How Oracle Processes Queries

By default, Oracle uses CBO and the default goal of CBO is increasing the *throughput*. If needed, the goal of CBO can be set to minimizing the *response time*.

The default behavior of the Oracle optimizer is defined by the initialization parameter OPTIMIZER_MODE, which can have one of the following values:

- CHOOSE. The optimizer chooses between CBO and RBO. If statistics are available for at least one of the tables involved in the query, the optimizer will use CBO. If no statistics are available for the tables of the query, then the RBO approach will be used. This is the default value of OPTIMIZER_MODE.
- ALL_ROWS. The optimizer uses CBO regardless of the presence of the statistics. The goal of the optimization is increasing the throughput.
- FIRST_ROWS. The optimizer uses CBO regardless of the presence of statistics. The goal of optimization is minimizing the response time.
- RULE. The optimizer uses RBO regardless of the presence of statistics.

This parameter can be defined for all sessions using the database or for a particular database session. For example, the following statement changes the default behavior of the optimizer to RBO for the current session only

```
ALTER SESSION SET OPTIMIZER_GOAL = RULE;
```

Hints

The application developer or user who understands the nature of the data and the database queries can suggest a better strategy than the one chosen by the optimizer in some cases. For example, if the optimizer has general statistics on the index on the deptType column for the table Department, but it does not have histograms on this column, then the optimizer may decide that the index is not selective enough and choose not to use it for processing of the query:

```
SELECT * FROM Department WHERE deptType = 'Marketing';
```

The application developer, on the other hand, knows that there are only a few departments with this department type and that using the index on the column can improve performance.

The developer can prompt the optimizer to choose another strategy with the help of hints. Hints are used to change the optimization approach, the goal of optimization, the type of access to data, and the way joins are processed. The hint affects the optimizer's behavior only for a particular query. Hints are inserted in the statement as comments with the sign '+'. For example, the hint RULE changes the approach of the processing of the previous statement from CBO to RBO, and it can force the optimizer to use the index on the column deptType:

```
SELECT /*+ RULE */ * FROM Department WHERE deptType = 'Marketing';
```

The developer can explicitly specify the index to be used (assume that the name of the index on the column deptType for the table Department is i0):

```
SELECT /*+ INDEX (Department, i0) */ * FROM Department WHERE deptType =
'Business';
```

The following example shows how to specify the order of joining tables in a query. For example, in the following query it is beneficial to join the selected rows of the table Department with the table Employee, and then join the result with the table Title. The hint prompts the optimizer to join tables in the order in which they are mentioned in the query:

```
SELECT /*+ ORDERED */ e.ID, e.name, t.salary
FROM Department d, Employee e, Title t
WHERE d.deptCode=e.deptCode AND e.titleCode=t.titleCode AND
   d.deptType = 'IT';
```

Oracle has more than 20 hints.

Monitoring

The previous section discussed how to change the optimizer's strategy for query executions. To do this, the developer has to be able to see how the optimizer executes a query.

The EXPLAIN PLAN statement displays the execution strategy for a query. Strategies are stored in the table PLAN_TABLE.The main columns of interest of this table are:

- OPERATION. The name of the operation performed, e.g. TABLE ACCESS.
- OPTIONS. The name of the operation (options) associated with the OPERATION, e.g. FULL.
- OBJECT_NAME. The name of the object of the operation, e.g. Department.

- ID. The number of the step.
- COST. The estimated cost of execution of the operation (NULL when RBO is used).
- CARDINALITY. The estimated number of rows accessed by the operation.
- STATEMENT_ID. Identifies the rows of PLAN_TABLE that refer to a particular query.

The following statement writes the execution strategy into the table PLAN_TABLE:

```
EXPLAIN PLAN SET STATEMENT_ID = 'P1' FOR
SELECT * FROM Department
WHERE location = 'Cleveland';
```

We can see the execution plan by selecting from PLAN_TABLE. The following statement selects from the PLAN_TABLE the estimated cost, operations with options, and objects of operations and formats the output:

```
SELECT ID || ' ' || PARENT_ID || ' ' ||
           LPAD(' ',2*(LEVEL - 1)) || OPERATION || ' ' || OPTIONS ||
           ' ' || OBJECT_NAME "Execution Plan"
FROM Plan_table
START WITH ID = 0 AND STATEMENT_ID = 'P1'
CONNECT BY PRIOR ID = PARENT_ID AND
           STATEMENT_ID = 'P1';
```

The result of this request shows that the query will be executed as a full-table scan:

```
Execution Plan
----------------------------------------------------
0        SELECT STATEMENT
1   0    TABLE ACCESS FULL DEPARTMENT
```

The *actual* performance of a database can be traced by turning the tracing of database performance on:

```
ALTER SESSION SET SQL_TRACE = true;
```

The trace file contains not only the execution strategies of queries, but also the detailed information about the consumption of resources. Using the generated trace file, the utility TKPROF produces a formatted output file that is used for performance analysis and for finding problems and bottlenecks.

The Performance of Distributed Queries

A query is distributed if it accesses data from a remote site. The local site (the site, which initiates the query) breaks the query into portions and sends them to remote sites for execution. The remote site executes its portion of the query and sends the result back to the local site. The local site then performs all the necessary post-processing and returns the results to the user or application.

The most effective way of optimizing the distributed query is to minimize the number of accesses to the remote databases and the amount of data retrieved from them.

Let us discuss a query on the distributed design of the Manufacturing Company case from Chapter 3. The query is originated in the New York database and has to return the names of those New York employees, whose salary is more than $20000 (the table Title is located in the Boston database):

```
SELECT name
FROM Employee e, Title@boston t
WHERE e.titleCode = t.titleCode AND
      t.salary > 20000;
```

To execute the query, for each row of the local fragment of Employee, the local site (New York) will access the remote site (Boston) to get the salary of the corresponding title. For each of these multiple trips to the remote site, the table Title will be searched for the corresponding title. A more efficient way to execute the query would be to access the remote site once, retrieve only the required rows, and then use these rows at the local site to produce the result.

In order to "prompt" Oracle on how to work with the remote site more efficiently and to optimize the execution of distributed queries, it is recommended to use collocated inline views (two or more tables of a query located in the same database are called collocated). The purpose of the collocated inline view is to define within the query a subquery that needs data only from one remote site. The conditions on the remote tables must be included in the view. The above distributed query can be rewritten with the inline view:

```
SELECT name
FROM Employee e,
      (SELECT titleCode
      FROM Title@boston
      WHERE salary > 20000) t
WHERE e.titleCode = t.titleCode;
```

The Boston site and the table Title there will be accessed once. Only the required rows of the table Title (with salary more than 20000) will be transferred to the New York database where the query processing will be completed.

Oracle's optimizer can transparently rewrite some distributed queries to take advantage of the performance gains offered by collocated inline views. However, you should remember that the form of the query still plays an important role in the query's execution.

Examples: Tuning Query Processing

Manufacturing Company Case

Let us consider tuning the centralized database for the Manufacturing Company case. Assume that the company has several hundred departments and several hundred thousand employees. Storage solutions for the case were suggested in Chapter 2. Here, we will discuss the usefulness of some indices.

According to the case description, users of each city need to work with data about local departments and employees of these departments. Access to local data (e.g. in Boston) is performed with the help of the queries:

```
SELECT * FROM Department WHERE location = 'Boston';

SELECT e.* FROM Employee e, Department d
WHERE e.deptCode = d.deptCode AND location = 'Boston';
```

The table Department is often accessed by queries with conditions … WHERE location = '?'. Such queries can benefit from an index on the column location:

```
CREATE INDEX i0 on Department(location);
```

Because the attribute location has only three values, we could consider a bitmap index instead of the B-tree index. However, the table Department is small (only several hundred rows), and Oracle will not use an index (of any kind), rather performing a full scan of the table. Therefore, the index is not needed.

The table Employee is involved in queries together with the table Department. For join queries, it is recommended to create the index on the foreign key of the child table, in our case, on the attribute deptCode of the table Employee:

```
CREATE INDEX i1 ON Employee(deptCode);
```

The application that requests data about employees of particular departments will also benefit from this index.

There is another application that processes data about employees of a specific type, for example full-time employees:

```
SELECT * FROM Employee WHERE emplType = 'Full-time';
```

To improve the performance of such queries, we will consider the index on the column emplType. Because this column has only three values, we choose the bitmap index:

```
CREATE BITMAP INDEX i2 ON Employee(emplType);
```

Analyzing Performance in Oracle

Appendix 5 shows a session of work in Oracle. Oracle execution strategies are analyzed for different queries that are executed on a table with several hundred thousand rows with and without indices, and with and without database statistics. The role of the database buffer is demonstrated.

Summary

The performance of queries is one of the main features, which define the quality of a database. Achieving good performance is a complicated problem, and its solution depends on the design, implementation, and management of the database.

Usually, the performance of queries is measured by response time and throughput. Minimizing the response time and maximizing throughput are different goals of performance management and they depend on the type of application.

Query processing consists of several steps: decomposition, optimization, and execution. Optimization is a very important part of query processing and it produces a strategy of query execution. The performance of the query depends on the strategy that is built by the optimizer. DBMSs apply two types of optimization: the heuristic or rule-based and cost-based. Heuristic optimization is based on rules of complexity of relational operations and does not take into consideration the current state of the database. Cost-based optimization estimates different strategies by calculating costs of their execution and chooses the cheapest strategy; to perform estimation database statistics are needed. For the same query, cost-based optimization may produce different strategies for different states of the database. Statistics are gathered by special utilities and are stored in the data dictionary.

Data design defines performance of the future database. The normalized data model is free from data modification anomalies and usually delivers better performance for modification queries. Retrieval of data on the normalized database, however, can suffer from bad performance because often multiple tables need to be joined. Denormalization of the database performed by merging tables is a common approach for improving the performance with the help of design. Storing very long columns in a separate table is another possibility that may be considered by designers. The distribution and localization of data can improve performance, especially for large databases.

The physical data design and implementation of the database largely define performance. The main goal of physical design is to minimize the number of disk accesses for query processing. Developers should decide how the data of each table are stored in data blocks and what type of storage is used for the table – heap or organized. Choosing the appropriate data types for columns of the table is also important for efficient query processing.

There are several ways to organize the storage of data. The most common approach is clustering – storing rows of the table that are often accessed together in the same data blocks. The index cluster stores together rows that have the same value of a particular attribute, which is called the cluster key. The index cluster has to be supported by an index on the cluster key. The hash cluster organizes rows with the same value of the hash function in the same blocks. Access to data in hash clusters is very efficient – the system calculates the value of the hash function and goes directly to the blocks with the requested data. However, while index clusters can facilitate queries with equality and range conditions, hash clusters are beneficial for queries with exact equality conditions only. Clusters should not be used for tables with intensive updating, inserting and deleting activity.

Data storage parameters that define where data has to be stored, how data is packed in data blocks, how the system maintains data, (and other parameters) have to be specified for each table, cluster and index.

Some DBMSs support other ways of organizing data storage. For example, in Oracle, tables can be partitioned or index-organized.

Access to data can be enforced by special objects – indices. Table indices are beneficial for selective read queries (which access not more than 25% of the rows of a table). Multi-column indices are involved in the processing of queries constrained by attributes of the left-most part of the index. Accessing indices only can be sufficient and, therefore, especially useful for the processing of queries that reference only indexed columns.

The successful design and implementation of a database do not guarantee good database performance. Performance depends on the database environment – capacity of the database server, and available memory and storage. Queries and applications on the database have to be implemented with knowledge of the query processing features of the DBMS and with an understanding of the database design and implementation.

Tuning a database starts with database design and continues into database maintenance:

- *Analyzing the nature of the data and the database requests*. Database professionals have to understand the purpose of the database, the nature of the data and the database requests.
- *Designing for performance*. Database design serves the purpose of the database and the nature of data and the database activities. The Physical design has to utilize the data storage capabilities of the DBMS. Consider

distributing data, clustering, partitioning, or other ways of organizing the data storage. Define the storage parameters of each table in correspondence with how the data of the table will be used.

- *Implementing for performance*. Configure the database environment and implement the database according to the design. Design and implement indices for important queries.
- *Tuning applications and queries*. Applications and queries have to utilize the data processing features of the DBMS and the design and implementation of the database.

Review Questions

1. What factors influence the performance of queries?
2. How is performance measured? Describe the relationships between different performance measures.
3. What factors affect the performance of distributed queries?
4. What are the steps of query processing and what is the purpose of each step?
5. Define what effective storage of data is?
6. What are the benefits of clustering? When does clustering have to be considered?
7. What are the different types of optimization? Under what circumstances are these different types of optimization used?
8. What are the prerequisites of cost-based optimization?
9. How does data design affect database performance?
10. What is the impact of the physical data model on performance?
11. How can application design improve the performance of the database?
12. Define the strategies of indexing.
13. How can composite indices improve the performance of some queries?
14. How can the database environment influence performance?
15. Why is the performance two hours after the start of a database better than immediately after the start?
16. How is data storage organized in Oracle? What types of storage organization are supported in Oracle?
17. What types of indices are supported in Oracle? In what situations is each index type beneficial?
18. What database statistics does the Oracle optimizer use? How can histograms improve performance?
19. What is the role of hints in Oracle queries?
20. How can one learn about Oracle's strategy for execution of a particular query?
21. Will indexed access be beneficial for a query that accesses 70% of rows of the table?

Practical Assignments

1. For the centralized database design for the Manufacturing Company case:
    a. Define the types of storage and storage parameters of the tables. Explain any assumptions and decisions you had to make. .
    b. Build indices for improving the performance of the mentioned applications. Explain any assumptions and decisions you had to make..
    c. Build queries for the mentioned requests in Oracle SQL.
    d. Check the execution plans for the queries for RBO and CBO.

2. Perform the tasks of the previous assignment for the other case studies.

3. Several queries are executed on the Manufacturing Company database. One query accesses the data about employees of a particular employee type and a particular department, another – about the employees of a particular employee type. Which index may improve performance of both queries?
    a. Composite index on emplType, deptCode.
    b. Composite index on ID, emplType, deptCode.
    c. Composite index on deptCode, emplType.
    d. Single-column index on emplType.

4. Describe the conditions under which clustering the table T(A, B, C, D) with the column C as the cluster key is beneficial.

5. Explain when you may consider adding the table S (X, Y, Z) to the cluster of the previous assignment.

6. On the table T(A, B, C, D) we have several queries. One query accesses data from T given the values of the attributes B and D, another – given the value of D, and the third query – given the value of the attribute C. What indices on the table T can improve the performance of the queries?

7. Partition the table T(A, B, C, D).
    a. Explain any assumptions you had to make.
    b. Write queries that will benefit from the partitioning.

8. For the distributed design of the Manufacturing Company database, build a query that returns data about all employees of a particular title (the query is executed from New York). Rewrite the query using inline collocated views and explain why the new query will be more efficient.

9. Build reduced views for the following requests on the Manufacturing Company database (the requests are executed from New York). When writing the queries, use inline collocated views where appropriate.
    a. Retrieve data about employees of marketing departments of Boston and New York.
    b. Retrieve data about Boston and Cleveland employees with titles T1 and T2.

10. The table T(A, B, C, D) has a B-tree index on the column B. Which of the following requests may benefit from the index? Where necessary, define additional conditions under which the index will be beneficial. If the query cannot benefit from the index, suggest other solutions for improving its performance.

```
a. SELECT * FROM T WHERE B = x;
b. SELECT * FROM T WHERE B <> x;
c. SELECT * FROM T WHERE B LIKE 'x%';
d. SELECT * FROM T WHERE SUBSTR (B, 1, 1) = 'x';
```

11. Assume that the table Employee of the Manufacturing Company case includes the column gender.
    a. Describe situations when using the index on the column will be beneficial.
    b. What type of index would you will recommend?
    c. Explain when the index will be beneficial if 95% of employees of the company are men, and 5% are women.

12. For a table, e.g. Employee of the Manufacturing Company case, describe data access scenarios and conditions under which you would organize the storage of the table as:
    a. Cluster.
    b. Partition.
    c. Index-organized.
    d. Distributed.