Programs Offered

Post Graduate Programmes (PG)

- Master of Business Administration
- Master of Computer Applications
- Master of Commerce (Financial Management / Financial Technology)
- Master of Arts (Journalism and Mass Communication)
- Master of Arts (Economics)
- Master of Arts (Public Policy and Governance)
- Master of Social Work
- Master of Arts (English)
- Master of Science (Information Technology) (ODL)
- Master of Science (Environmental Science) (ODL)

Diploma Programmes

- Post Graduate Diploma (Management)
- Post Graduate Diploma (Logistics)
- Post Graduate Diploma (Machine Learning and Artificial Intelligence)
- Post Graduate Diploma (Data Science)

Undergraduate Programmes (UG)

- Bachelor of Business Administration
- Bachelor of Computer Applications
- Bachelor of Commerce
- Bachelor of Arts (Journalism and Mass Communication)
- Bachelor of Arts (General / Political Science / Economics / English / Sociology)
- Bachelor of Social Work
- Bachelor of Science (Information Technology) (ODL)

AMITY UNIVERSITY ONLINE
Amity Helpline: (Toll free) 18001023434
For Student Support: +91 - 8826334455
Support Email id: studentsupport@amityonline.com | https://amityonline.com

Product code
81030112000594

Advanced Database Management Systems

AMITY

Advanced Database Management Systems

AMITY UNIVERSITY ONLINE

Advanced Database Management Systems

AMITY UNIVERSITY ONLINE

Published by Amity University Press for exclusive use of Amity Directorate of Distance and Online Education, Amity University, Noida-201313

Contents

Module -I: Query Processing

Notes

Learning Objectives

At the end of this module, you will be able to:

- Understand basic concepts of query processing
- Define parsing and translation
- Infer query tree and query graph
- Discuss functional dependencies and normal forms
- Analyse heuristic optimisation of query tree

Introduction

The primary goal of query processing is to convert a high-level query expressed in Structured Query Language into a low-level query that implements relational algebra in an effective, accurate and sequential manner to retrieve the required data from the database. Choosing an effective execution strategy for a query is known as query optimisation and it is one of the most crucial aspects of query processing. There are four main phases of query processing.

- ❖ Decomposition
- ❖ Optimisation
- ❖ Code generation
- ❖ Execution.

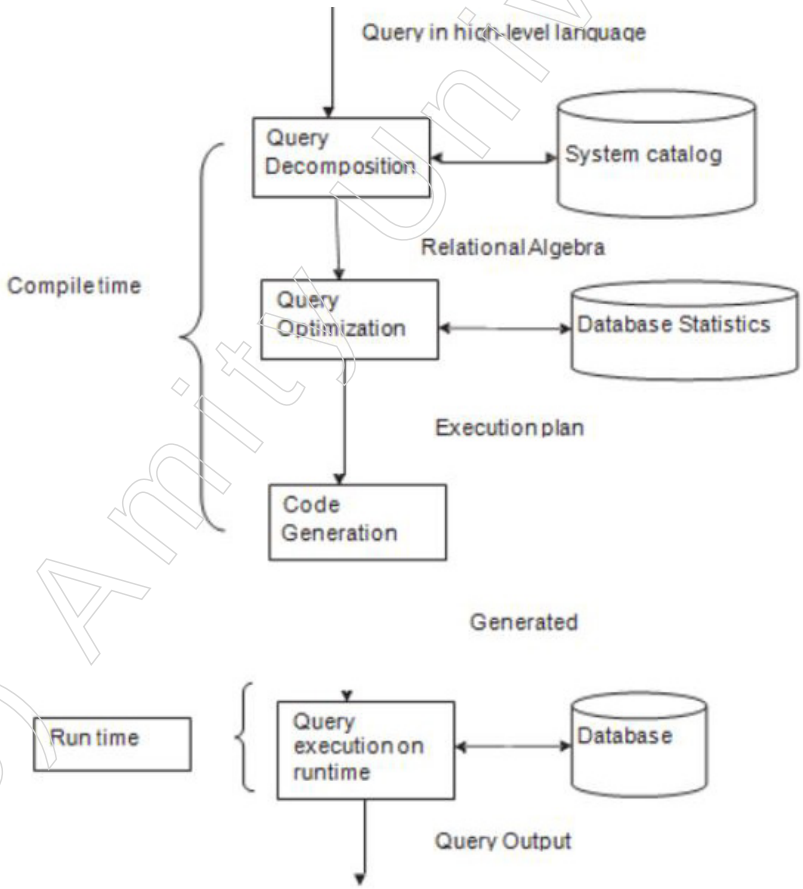


Figure: Query Processing Procedure

Notes

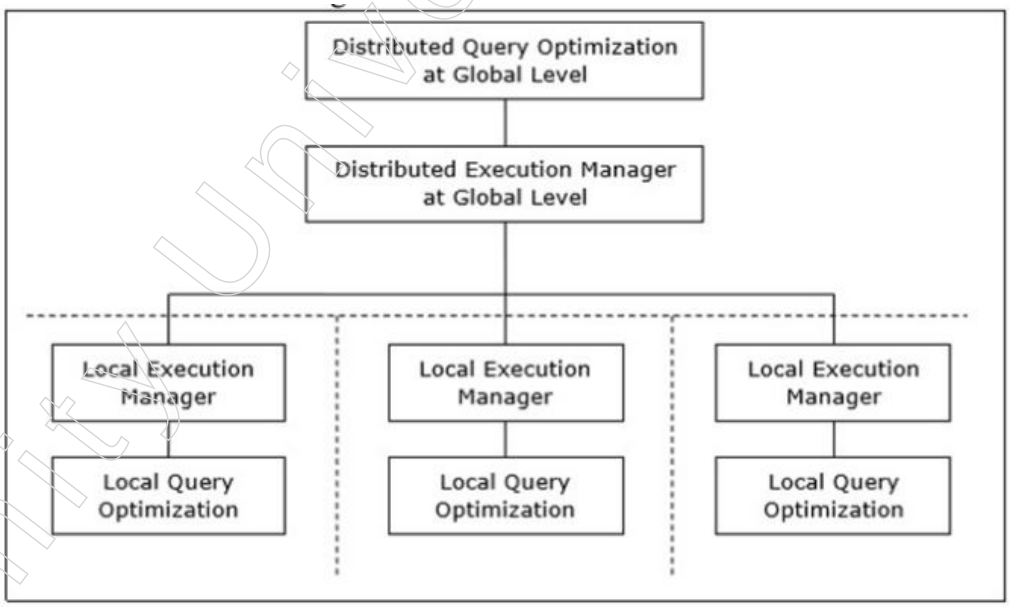
The basic aim of storing data in a database is to be able to query it. A mapping from a database instance D to a relation q(D) of fixed arity is all that a query q is. The variety of operations involved in obtaining data from a database is referred to as query processing. The tasks include query optimisation, query evaluation and translation of queries written in high-level database languages into expressions that may be employed at the file system's physical level.

1.1 Overview of Query Processing and Optimisation in Databases

In a distributed database management system, data must be transmitted between computers connected to a network to perform queries. In a database system, the ordering of data diffusions and local data processing is a distribution technique for a query. A distributed database management system query typically requires data from several sites; this requirement for data from various sites is known as the transmission of data that results in communication expenses.

Because data transmission via a network has a communication cost, query processing in centralised DBMS differs from query processing in DBMS. When sites are connected via high-speed networks, the transmission cost is minimal, but in other networks, it can be rather large.

In a distributed database system, responding to a query involves both global and local optimisation. The query is moved to the client or supervisory site's database system. In this case, the user is legalised and the query is verified, translated and improved globally. One way to interpret the architecture is



Mapping Global Queries into Local Queries

The process that converts global requests into local ones is identifiable as follows:

- The tables that are necessary for a global query are fragmented among several sites. There is very little data in the local databases. Utilising the global data dictionary, the supervisory site gathers distribution-related data and assembles the pieces to reconstruct the global vision.
- The global optimiser monitors local queries at the locations where the fragments are stored if there is no duplication. The global optimiser chooses the site based on workload, server speed and communication cost if replication is present.

Notes

- The distributed execution proposal generated by the global optimiser ensures that the least amount of data is distributed throughout the sites. The plan determines the placement of the pieces, the desired sequence for executing query stages and the procedures for transferring transitional results.
- The local database servers optimise the local queries. In the end, the local query impacts are combined using a join procedure for vertical fragments and a blending operation for horizontal fragments.

Query Optimisation

Evaluation of a massive number of query trees, each of which yields the required results of a query, is required for distributed query optimisation. The main cause of this is the prevalence of widely replicated and fragmented data. Therefore, finding an optimal solution rather than the best solution is the aim.

When it comes to distributed query optimisation, the primary issues are

- ❖ Optimal consumption of resources in the distributed system.
- ❖ Query trading.
- ❖ Decrease of solution space of the query.

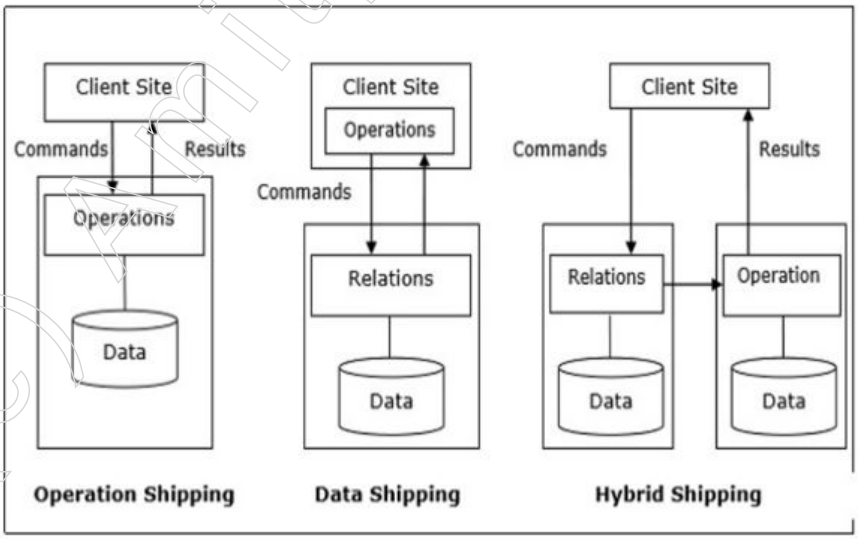
Optimal Utilisation of Resources in the Distributed System

Many database servers located at different locations execute the tasks associated with a query in a distributed system. The methods for making the best use of available resources are as follows:

Operation Shipping: Instead of being carried out at the client site, operation shipping is carried out in the location where the data is stored. After that, the results are delivered to the customer location. This is relevant to operations in which the operands are available at the same location. e.g., Choose and Project activities.

Data Shipping: Fact fragments are transferred to the database server via data shipping, where they are processed. This is applied to processes where the operands are dispersed across multiple locations. This works well in systems with low communication overheads and many local processors that operate at a slower speed than the client-server ones.

Hybrid Shipping: This is a hybrid of operation shipping and data shipping. Data fragments are now sent to the fast processors, where the process is executed. The customer site is then reached from the results.



Notes

1.1.1 Basic Concepts of Query Processing

Processing a query can include several different approaches, particularly if it is complex. Nonetheless, it is typically beneficial for the system to invest a significant amount of effort in choosing a method. Usually, information from main memory can be used to determine a strategy with little to no disc access.

There will be multiple disc accesses during the query's actual execution. Saving disc accesses requires a significant amount of computing power because disc data transport is slower than that of main memory and the computer system's central CPU.

Basic steps in Query Processing

The methodical process of converting high-level language into low-level language that a machine can comprehend and use to carry out a user's request is known as query processing. This query processing is done by the DBMS's query processor. The following are the steps in the processing process:

1. Parsing and translation.
2. Optimisation.
3. Evaluation
4. Execution

The system has to convert the query into a form that can be used before query processing can start. While a language like SQL is fine for human usage, it is not the best choice for representing a query inside within a system. An internal representation based on the extended relational algebra is more practical. In order to process a query, the system must first translate the given query into its internal form.

This translation procedure is comparable to the tasks carried out by a compiler's parser. The parser analyses the user's query syntax, confirms that the relation names appearing in the query are names of the relations in the database and so on before generating the internal form of the query.

The question is first represented as a parse-tree, which the system then converts into a relational-algebra expression. In case the query was stated using a view, the relational-algebra expression that defines the view will take the place of the view in all instances during the translation process. The image below illustrates the fundamental steps:

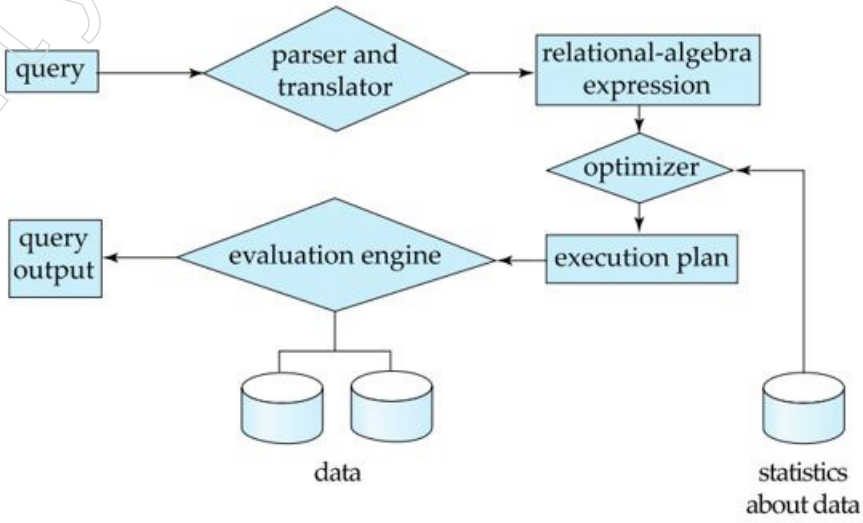


Figure: Stages of query processing

Notes

There are usually several approaches to compute the answer to a given query. For instance, we've seen that a query in SQL can be written in multiple ways.

There are multiple techniques to convert each SQL query into a relational algebra expression. Furthermore, there are typically multiple ways to evaluate relational algebra expressions; the relational-algebra form of a query only partially describes how to evaluate a query.

An evaluation primitive is a relation algebra operation annotated with evaluation guidelines. A query-execution plan, also known as a query-evaluation plan, is a series of basic actions that can be used to assess a query. The query execution engine receives a query-evaluation plan, carries it out and outputs the query's responses.

The charges of the various assessment plans for a particular query may vary. We do not anticipate that users will design their questions in a way that recommends the best course of action for evaluation. Instead, the system must create a plan for query evaluation that minimises query evaluation costs; this process is known as query optimisation.

After selecting a query plan, the query is assessed using that plan and the query's outcome is output. The following is a summary of the primary steps:

1. Parsing and Translating: The query processor selects any query sent to the database first. The query is scanned, parsed into individual tokens and then checked to make sure it is correct. It verifies the query's syntax and the accuracy of the tables and views that are being utilised. Each token is then transformed into relational expressions, trees and graphs after it has passed. The other parsers in the DBMS can handle these with ease.
2. Evaluation: A physical query plan, also known as an execution plan, is sent to the query execution engine, which then carries it out and gives the outcome.

   The following is the order in which query evaluation is completed:

   - Cartesian products are used to combine the tables in the from clause.
   - Next, the where predicate is used.
   - The tuples that are produced are then grouped by group by clause.
   - Every group is subject to the having predicate, which may result in the exclusion of some groupings.
   - Every last group is subjected to the aggregates. Finally, the select clause is executed.
3. Optimisation: Find the query's most affordable execution strategy.

1.1.2 Converting SQL Queries into Relational Algebra

A data model requires a mechanism to query and alter the data in addition to its structure. We will start operations on relations by learning about a unique algebra known as relational algebra, which consists of a few straightforward yet effective methods for creating new relations out of existing ones. The created relations can provide responses to questions regarding the stored data if the specified relations represent data that has been stored.

Relational algebra is not utilised nowadays as a query language in commercial DBMS's, however some of the early prototypes did use relational algebra directly. Instead, relational algebra lies at the core of the "real" query language, SQL and many SQL programs are essentially "syntactically sugared" formulations of relational algebra. Additionally, a SQL query is first transformed into relational algebra or a very comparable internal representation by a DBMS when it processes the query.

Notes

Prior to going over relational algebra operations, it is important to consider whether or not we actually need a new class of database programming languages. Can't every computable question regarding relations be asked and answered in traditional languages like C or Java? In the end, a tuple of a relation can be represented by an object (in Java) or a struct (in C) and relations can be represented by arrays of these elements.

The unexpected response is that because relational algebra is not as powerful as C or Java, it can still be useful. In other words, relational algebra does not allow one to execute certain computations that can be done in any conventional language. Determining whether a relation's tuple count is even or odd is one example. We gain two major benefits from restricting what we may say or do in our query language: programming is made easier and the compiler is able to generate highly optimised code.

Overview of Relational Algebra

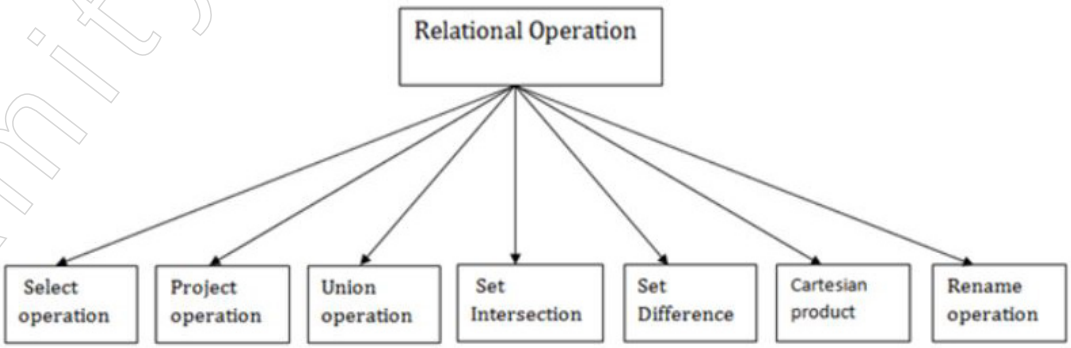
Four major classes comprise the operations of the classical relational algebra:

1. The standard set operations (union, intersection and difference) applied to relations;
2. operations that remove parts of a relation: "selection" removes some rows (tuples);
3. "projection" removes some columns; operations that combine the tuples of two relations, such as "Cartesian product," which pairs the tuples of two relations in every possible way; and
4. various types of "join" operations, which pair tuples from two relations in a selective manner;
5. an operation known as "renaming" that modifies the relation schema but does not affect the tuples of a relation.

A procedural query language known as relational algebra accepts relation instances as input and produces relation instances as output. It uses operators to help it execute queries. You can use a unary or binary operator. As inputs, they receive relations and as outputs, they make relations. A relationship is treated as a recursive relational algebra, with intermediate results being regarded as relations as well.

Procedural algebra is a language for creating queries. It provides a step-by-step guide on how to get the query's result. To execute queries, it makes use of operators.

Types of Relational operation



1. Select Operation:
   - ❖ Tuples that meet a certain predicate are chosen using the select operation.
   - ❖ The symbol for it is sigma (σ).

   Notation: σ p(r)

Notes

Where:

σ is used for selection prediction

r is used for relation

P is a formula for propositional logic that can include connectors like AND, OR and NOT. These relational can utilise as relational operators like =, ≠, ≥, <, >, ≤.

2. Project Operation: The list of qualities that we want to see in the outcome is displayed by this operation. The remaining properties have been removed from the table.

   It is denoted by ∏.

   Notation: ∏ A1, A2, An (r)

   Where

   Relation r's attribute name is A1, A2 and A3.

3. Union Operation:

   Assume that R and S are two tuples. All of the tuples that are either in R or S or both in R & S are contained in the union operation.

   The redundant tuples are removed. ∪ is used to indicate it.

   Notation: R ∪ S

   The following requirements must be met by a union operation:

   - ❖ The same number must be an attribute shared by R and S.
   - ❖ Tuple duplicates are automatically removed.

4. Set Intersection:

   - ❖ Assume that R and S are two tuples. Every tuple that is in both R and S is contained in the set intersection operation.
   - ❖ The symbol for it is intersection ∩.

   Notation: R ∩ S

5. Set Difference:

   - ❖ Assume that R and S are two tuples. All tuples that are in R but not in S are contained in the set intersection operation.
   - ❖ The symbol for it is intersection minus (-).

   Notation: R - S

6. Cartesian Product:

   - ❖ Every row in one table is combined with every row in the other table using the Cartesian product. Another name for it is a cross product.
   - ❖ X is used to represent it.

   Notation: E X D

7. Rename Operation:

   - ❖ The output relation is renamed using the rename operation. The symbol for it is rho (ρ).

   ρ(STUDENT1, STUDENT)

To access and modify data in databases, one uses SQL, a standard database language. Structured Query Language is known as SQL. In the 1970s, IBM computer

Notes

scientists invented SQL. SQL can create, edit, remove and retrieve data from databases such as PostgreSQL, Oracle, MySQL and others by running queries. All things considered, SQL is a query language used to interact with databases.

A common database language used to build, manage and access relational databases is called structured query language. We shall go into great detail regarding SQL in this tutorial. Here are a few intriguing SQL-related facts. Let's concentrate on that.

SQL does not care about case. However, it is best practice to use capital letters for keywords (such as SELECT, UPDATE, CREATE, etc.) and tiny letters for user-defined items (such as table names and column names).

In SQL, comments can be written at the start of any line using the double hyphen "–". SQL is the computer language used by relational databases such as Postgre, MySQL, Oracle, Sybase and others (described below). SQL is not used by other non-relational databases (also known as NoSQL databases), such as MongoDB, DynamoDB, etc.

While SQL has an ISO standard, the majority of implementations have slightly different syntax. As a result, there may be queries that execute well in SQL Server but not in MySQL.

How Queries can be categorised in Relational Database?

Relational database queries fall into one of two categories:

- Data Definition Language: This language is used to specify the database's structure. CREATE TABLE, ADD COLUMN, DROP COLUMN and so forth are some examples.
- Data Manipulation Language: This language is employed to modify data within relationships. e.g.; INSERT, DELETE, UPDATE and so on.
- Data Query Language: This language is used to retrieve data from relationships. for instance, SELECT Thus, we'll start by talking about the Data Query Language. To obtain data from a relational database, a general query to use is:
  1. SELECT [DISTINCT] Attribute_List FROM R1,R2….RM
  2. [WHERE condition]
  3. [GROUP BY (Attributes)[HAVING condition]]
  4. [ORDER BY(Attributes)[DESC]];

You must execute at least a portion of the query shown in statement 1 in order to obtain data from a relational database. The statements enclosed in [] are not required.

Converting SQL Queries into Relational Algebra

In actuality, the majority of commercial RDBMSs employ SQL as their query language. An optimised extended relational algebra expression, which is represented as a query tree data structure, is first created by translating a SQL query.

SQL queries are generally broken down into query blocks, which are the fundamental building blocks that can be optimised and converted into algebraic operators. A query block includes GROUP BY and HAVING clauses, if present, in addition to a single SELECT-FROM-WHERE phrase. Therefore, queries that are nested inside other inquiries are recognised as distinct query blocks. The aggregation operators in SQL, such MAX, MIN, SUM and COUNT, need to be included in the extended algebra as well.

Translating SQL statements into the formal language of relational algebra is the first step in converting SQL queries to expressions in relational algebra. A mathematical query language called relational algebra works with relations, which are the relational database's

equivalent of tables. The essential procedures and guidelines for translating SQL queries into relational algebra are as follows:

Notes

1. Select Operation:

SQL: SELECT ... FROM ... WHERE ...

Relational Algebra: σ_condition(π_columns(relation))

Example:

SELECT name, age FROM Students WHERE grade = 'A';

Relational Algebra:

π_{name, age}(σ_{grade='A'}(Students))

2. Project Operation:

SQL: SELECT ... FROM ...

Relational Algebra: π_columns(relation)

Example:

SELECT name, age FROM Students;

Relational Algebra:

π_{name, age}(Students)

3. Join Operation:

SQL: SELECT ... FROM table1 INNER JOIN table2 ON condition

Relational Algebra: relation1⋈_{condition} relation2

Example:

SELECT Students.name, Courses.title FROM Students INNER JOIN Enrollments ON Students.id = Enrollments.student_id INNER JOIN Courses ON Enrollments.course_id = Courses.id;

Relational Algebra:

Students ⋈_{Students.id = Enrollments.student_id} Enrollments ⋈_{Enrollments.course_id = Courses.id} Courses

4. Union Operation:

SQL: SELECT ... FROM table1 UNION SELECT ... FROM table2

Relational Algebra: relation1 ⋈ relation2

Example:

SELECT name, age FROM Students WHERE grade = 'A' UNION SELECT name, age FROM Students WHERE grade = 'B';

Relational Algebra:

π_{name, age}(σ_{grade='A'}(Students)) ⋈ π_{name, age}(σ_{grade='B'}(Students))

5. Intersection Operation:

Notes

SQL: SELECT ... FROM table1 INTERSECT SELECT ... FROM table2

Relational Algebra: relation1 ∩ relation2

Example:

SELECT name, age FROM Students WHERE grade = 'A' INTERSECT SELECT name, age FROM Students WHERE grade = 'B';

Relational Algebra:

π_{name, age}(σ_{grade='A'}(Students)) ∩ π_{name, age}(σ_{grade='B'}(Students))

6. Difference Operation:

SQL: SELECT ... FROM table1 EXCEPT SELECT ... FROM table2

Relational Algebra: relation1 - relation2

Example:

SELECT name, age FROM Students WHERE grade = 'A' EXCEPT SELECT name, age FROM Students WHERE grade = 'B';

Relational Algebra:

π_{name, age}(σ_{grade='A'}(Students)) - π_{name, age}(σ_{grade='B'}(Students))

A few fundamental SQL queries and the related relational algebra expressions are covered in these examples. More intricate SQL searches might have aggregation functions or nested subqueries, in which case further steps and thought would need to be taken while translating them to relational algebra. It's also vital to keep in mind that relational algebra and SQL differ somewhat in terms of expressive capacity, meaning that not all SQL queries can be translated directly into relational algebra.

1.1.3 Basic Algorithms for Executing Query Operations

The relational algebra is the fundamental set of operations for the formal relational model. A user can provide simple retrieval requests as relational algebra expressions using these procedures. The outcome of a retrieval query is a new relation. As a result, the algebraic operations generate new relations that can be further worked with by other algebraic operations. A relational algebra expression is created by a series of relational algebra operations and the expression's outcome is a relation that symbolises the outcome of a database query.

There are various reasons why the relational algebra is crucial.

- it offers a structured basis for relational model functions.
- In the query processing and optimisation modules, which are essential components of relational database management systems (RDBMSs), it serves as the foundation for implementing and optimising queries.
- For RDBMSs, the SQL standard query language incorporates some of its ideas. The fundamental operations and functions in the internal modules of the majority of relational systems are based on relational algebra operations, despite the fact that the majority of commercial RDBMSs in use today do not offer user interfaces for relational algebra queries.

The SELECT Operation

To choose a subset of tuples from a relation that meets a selection condition, use the

Notes

choose operation. The SELECT function can be thought of as a filter that retains only tuples that meet a qualifying requirement. As an alternative, we may think of using the SELECT operation to limit the set of tuples to just those that meet the criteria.

One way to represent the SELECT operation is as a horizontal division of the relation into two sets of tuples: the ones that meet the requirement and are chosen and the ones that don't meet the condition and are eliminated. For instance, using a choose operation, we may independently define each of these two requirements to choose the EMPLOYEE tuples whose department is 4 or those whose income is greater than $30,000:

$\sigma^{Dno=4}$(EMPLOYEE)

$\sigma^{Salary>30000}$(EMPLOYEE)

In general, the SELECT operation is denoted by

$\sigma^{<selection\ condition>}$(R)

where the selection condition is a Boolean expression (condition) defined on the attributes of relation R and the SELECT operator is represented by the symbol σ (sigma). Observe that R is often an expression in relational algebra that returns a relation; the most basic example of this type of expression is just the name of a relation in a database. The properties of R are also present in the relation that emerges from the SELECT action.

The Boolean statement given in consists of several clauses of the following type.

< attribute name>< comparison op>< constant value>

Or

<attribute name><comparison op><attribute name>

where <attribute name> is the name of an attribute of R, <comparison op> is generally one of the operators {=, <, ≤, >, ≥, ≠} and <constant value> is a constant value from the attribute domain. Clauses do not need to form a general selection condition in order to be joined by the ordinary Boolean operators and, or. For instance, we can use the following SELECT operation to choose the tuples for all workers who either make over $25,000 annually in department 4 or over $30,000 annually in department 5:

σ(Dno=4 AND Salary>25000) OR (Dno=5 AND Salary>30000)(EMPLOYEE)

The result is shown in Figure below

(a)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

(b)

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Figure: Results of SELECT and PROJECT operations. (a) σ(Dno=4 AND Salary>25000) OR (Dno=5 AND Salary>30000) (EMPLOYEE).(b) πLname, Fname, Salary(EMPLOYEE). (c) πSex, Salary(EMPLOYEE).

Notes

Note that characteristics whose domains are ordered values, like date or numeric domains, can apply to all comparison operators in the set {=, <, ≤, >, ≥, ≠}. The collating sequence of the characters is also taken into consideration when ordering domains of character strings.

Only the comparison operators in the set {=, ≠} can be applied when an attribute's domain is a collection of unordered values. The domain Colour = {'red','blue','green','white','yellow',..} is an example of an unordered domain since there is no specified order for the different colours. Additional comparison operators are permitted in some domains; for instance, the comparison operator SUBSTRING_OF may be permitted in a domain containing character strings.

Generally speaking, one can ascertain the outcome of a SELECT operation in this manner. Every tuple t in R receives an independent application of the <selection condition>. In order to accomplish this, each time an attribute Ai appears in the selection condition, its value from the tuple t[Ai] is substituted. Tuple t is chosen if the condition evaluates to TRUE. The outcome of the SELECT operation includes each of the selected tuples. The following is the standard understanding of the Boolean conditions AND, OR and NOT:

- (cond1 AND cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
- (cond1 OR cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
- (NOT cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

The SELECT operator is applied to a single relation and is hence unary. Furthermore, since each tuple is subjected to the selection procedure separately, selection conditions are limited to one tuple. The number of attributes that make up the degree of the relation that results from a SELECT operation is equal to the degree of R. There is never more than or equal to the number of tuples in R in the resultant relation. That is, $|\sigma c (R)| \leq |R|$ for every condition C. The selectivity of a selection condition is defined as the percentage of tuples that the condition selects.

The PROJECT Operation

When we do a SELECT operation on a relational table, some rows are selected while other rows are discarded. In contrast, the PROJECT procedure picks out specific columns from the database and eliminates the others. The PROJECT procedure is used to project a relation across some qualities exclusively, if we are only interested in some of these attributes.

As a result, the PROJECT operation's output can be seen as a vertical division of the relation into two relations, one containing the operation's output and the necessary columns (attributes) and the other including the columns that were discarded. For instance, we can use the PROJECT operation as follows to list the first and last name as well as the pay of each employee:

$$\pi_{Lname, Fname, Salary}(EMPLOYEE)$$

The relation that results is displayed in Figure (b) above. The PROJECT operation's typical format is

$$\pi_{<attribute\ list>}(R)$$

where <attribute list> is the desired sublist of attributes from the attributes of relation R and π (pi) is the symbol used to symbolise the PROJECT operation. Once more, note that R is, in general, an expression in relational algebra whose output is a relation, which is, in

Notes

the most basic scenario, merely the name of a relation in a database. Only the properties listed in <attribute list>, in the same order as they appear in the list, are present in the outcome of the PROJECT action. As a result, the number of qualities in <attribute list> equals its degree.

Duplicate tuples are likely to arise if the attribute list only consists of R's nonkey attributes. The PROJECT operation yields a set of distinct tuples and, as a result, a valid relation because it eliminates any duplicate tuples. We call this process "duplicate elimination." Take the following PROJECT operation, for instance:

$\pi_{Sex, Salary}(EMPLOYEE)$

The outcome is displayed in Figure (c) above. Despite the fact that this set of data appears twice in the EMPLOYEE relation, the tuple <'F', 25000> only shows once in Figure 8.1(c). Sorting or using another method to find duplicates is known as duplicate elimination, which increases processing. Instead of a set, the outcome would be a multiset or bag of tuples if duplicates are not removed. In the formal relational model, this was not allowed; however, SQL permits this.

A PROJECT operation always produces a relation with a tuple count that is less than or equal to the tuple count in R. The resulting relation has the same number of tuples as R if the projection list contains some key of R, or if it is a superkey of R. Additionally,

$\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$

if the attributes in <list1> are present in <list2>; if not, the phrase on the left is wrong. Notably, commutativity is not maintained for PROJECT.

The SELECT clause of a SQL query contains the PROJECT attribute list. For instance, the subsequent procedure:

$\pi^{Sex, Salary}(EMPLOYEE)$

would be equivalent to the SQL query that follows:

```
SELECT DISTINCT Sex, Salary
FROM EMPLOYEE
```

You'll see that duplicates won't be removed from this SQL query if the keyword DISTINCT is removed. Although the formal relational algebra does not provide this option, it can be expanded to allow relations to be multisets and contain this operation; we will not go into these changes here.

The JOIN Operation

To merge related tuples from two relations into a single "longer" tuple, utilise the JOIN procedure, represented by ⋈. Because it enables us to handle relationships among relations, this function is crucial for any relational database that has more than one relation. As an example of JOIN, let's say we wish to get the manager's name for every department. Each department tuple must be combined with the employee tuple whose Ssn value corresponds to the Mgr_ssn value in the department tuple in order to obtain the manager's name. In order to accomplish this, we first project the outcome across the required properties using the JOIN procedure, as seen below:

$DEPT\_MGR \leftarrow DEPARTMENT \bowtie^{Mgr\_ssn=Ssn} EMPLOYEE$

$RESULT \leftarrow \pi^{Dname, Lname, Fname}(DEPT\_MGR)$

The figure below shows the initial operation. It should be noted that Ssn, the

Notes

primary key of the EMPLOYEE relation, is referenced by Mgr_ssn, a foreign key of the DEPARTMENT relation. The referenced relation EMPLOYEE's matched tuples are a result of this referential integrity restriction.

DEPT_MGR

Dname	Dnumber	Mgr_ssn	···	Fname	Minit	Lname	Ssn	···
Research	5	333445555	···	Franklin	T	Wong	333445555	···
Administration	4	987654321	···	Jennifer	S	Wallace	987654321	···
Headquarters	1	888665555	···	James	E	Borg	888665555	···

Figure: Result of the JOIN operation DEPT_MGR ← DEPARTMENT ⋈Mgr_ssn=Ssn EMPLOYEE

A typical condition for a join is of the type

<condition> AND <condition> AND … AND <condition>

where θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠} and each <condition> is of the type Ai θ Bj, where Ai is an attribute of R, Bj is an attribute of S and Ai and Bj have the same domain. A THETA JOIN is a JOIN operation that has such a generic join condition. The result does not include tuples whose join attributes are NULL or whose join condition is FALSE. Because tuples that do not get joined with matching ones in the other relation do not appear in the result, the JOIN operation does not always preserve all of the information in the participating relations.

The UNION, INTERSECTION and MINUS Operations

The common mathematical operations on sets make up the next class of relational algebra operations. For instance, we may perform the UNION operation as follows to obtain the Social Security numbers of all employees who either directly supervise or work in department 5:

$$DEP5\_EMPS \leftarrow \sigma^{Dno=5}(EMPLOYEE)$$

$$RESULT1 \leftarrow \pi^{Ssn}(DEP5\_EMPS)$$

$$RESULT2(Ssn) \leftarrow \pi^{Super\_ssn}(DEP5\_EMPS)$$

$$RESULT \leftarrow RESULT1 \cup RESULT2$$

The Ssns of all workers who work in department 5 are stored in the relation RESULT1, whereas the SSNs of all employees who directly supervise an employee who works in department 5 are stored in RESULT2. After removing any duplicates, the UNION operation generates the tuples that are in either RESULT1 or RESULT2 or both (see Figure below). As a result, the Ssn value "333445555" only occurs once in the outcome.

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

Figure: Result of the UNION operation RESULT ← RESULT1 ∪ RESULT2

To combine the elements of two sets in different ways, a number of set theoretic operations are utilised, such as SET DIFFERENCE (also known as MINUS or EXCEPT), INTERSECTION and UNION. Since each of these actions is applied to two sets

Notes

(of tuples), they are binary in nature. The two relations on which any of these three operations is performed must have the same type of tuples; this requirement is known as union compatibility or type compatibility when these operations are applied to relational databases.

If dom(Ai) = dom(Bi) for 1 <i ≤ n and two relations R(A1, A2, …, An) and S(B1, B2, …, Bn) have the same degree n, they are considered union compatible, also known as type compatible. This indicates that each matching pair of attributes has the same domain and that the two relations have the same amount of attributes.

On two union-compatible relations, R and S, we can define the three operations, UNION, INTERSECTION and SET DIFFERENCE as follows:

UNION: This procedure yields a relation, represented by R ∪ S, which contains all tuples that are either in R, in S, or in both R and S. Tuple duplication is removed.

INTERSECTION: This operation yields a relation that contains all tuples that are in both R and S, which is indicated by the notation R ∩ S.

SET DIFFERENCE (or MINUS): This operation yields a relation that contains all tuples that are in R but not in S, which is indicated by the notation R – S.

The convention that the new relation's attribute names match those of the original relation R shall be followed. The rename operator can always be used to rename the characteristics in the result.

The three operations are depicted in the figure below. The relations STUDENT and INSTRUCTOR in Figure (a) are union compatible and their tuples represent the names of students and the names of instructors, respectively. All of the students' and teachers' names are displayed in Figure (b), which is the outcome of the UNION operation. Keep in mind that the result only contains duplicate tuples once. Only individuals who are both pupils and instructors are included in the outcome of the INTERSECTION operation Figure (c).

Observe that the operations UNION and INTERSECTION are commutative, meaning that

$R \cup S = S \cup R$ and $R \cap S = S \cap R$

Since both UNION and INTERSECTION are associative operations, they can be seen as n-ary operations that apply to any number of relations.

$R \cup (S \cup T) = (R \cup S) \cup T$ and $(R \cap S) \cap T = R \cap (S \cap T)$

Generally speaking, the MINUS operation is not commutative.

$R - S \neq S - R$

The names of students who are not teachers are displayed in Figure (d) and the names of instructors who are not students are displayed in Figure (e).

Keep in mind that the following is how INTERSECTION can be described in terms of union and set difference:

$R \cap S = ((R \cup S) - (R - S)) - (S - R)$

The set operations covered here are mapped to three SQL operations: UNION, INTERSECT and EXCEPT. Furthermore, duplicates are not removed by the multiset procedures UNION ALL, INTERSECT ALL and EXCEPT ALL.

Cartesian Product

The set of pairings that may be created by selecting any element from S for the second

Notes

element and any element from R for the first is known as the Cartesian product (also known as the cross-product or just product) of two sets, R and S. R x S is the product's designation. The product is nearly the same when R and S are relatives. The outcome of pairing a tuple from R with a tuple from S is a lengthier tuple, with one component for each of the components of the constituent tuples, because the members of R and S are tuples, which typically consist of more than one component. In the attribute order for the result, the components from R (the left operand) normally come before the components from S.

The relation schema for the resulting relation is the union of the schemas for R and S. On the other hand, we must come up with new names for at least one of each pair of identical qualities if R and S might chance to share certain attributes. We utilise R.A for the attribute from R and S in order to distinguish an attribute A that is present in both the R and the 5 schemas.Afor the quality derived from S.

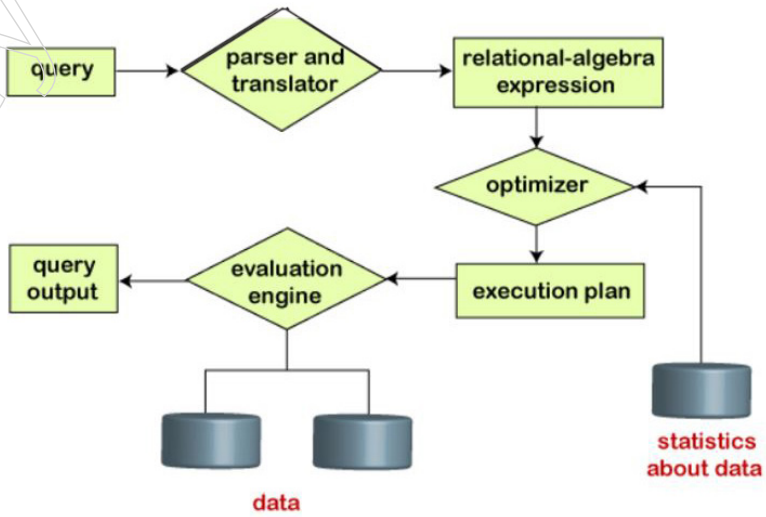
1.1.4 Parsing and Translation

Given that query processing involves some data retrieval tasks. The user requests are first converted into high-level database languages like SQL. It is converted into phrases that are usable at the file system's physical level. Following this, a number of query-optimising changes and the actual evaluation of the queries occur. A computer system must therefore convert a query into a language that is comprehensible and readable by humans before it can be processed.

Thus, the greatest option for humans is SQL, or Structured Query Language. However, it is not entirely appropriate for the query's internal representation within the system. The internal representation of a query is ideally suited for relational algebra. The translation process in query processing is comparable to the parser of a query. When a user runs a query, the system's parser generates the internal form of the query by first examining the query's syntax, then confirming the name of the relation in the database, the tuple and lastly the necessary attribute value.

The query is transformed into a "parse-tree" by the parser. Additionally, convert it to relational algebraic form. This effectively eliminates the need for any views in the query.

As a result, the diagram below helps us comprehend how a query processing system operates:



Let's say someone runs a query. As we now know, there are several ways to get the data out of the database. A user wishes to retrieve all employee records in SQL whose pay is at least $10,000. In order to accomplish this, the following inquiry is made:

Notes

select emp_name from Employee where salary>10000;

Therefore, the user question must be translated into relational algebraic form in order for the machine to comprehend it. This question can be expressed in relational algebraic form as follows:

$\sigma_{salary>10000}(\pi^{salary}(Employee))$

$\pi_{salary}(\sigma^{salary>10000}(Employee))$

We can use many algorithms to carry out each relational algebra operation after translating the provided query. Thus, a query processing starts to function in this manner.

1.1.5 Optimisation

The technique of enhancing the effectiveness and speed of SQL queries—which are used to access and modify data stored in relational database management systems (RDBMS)—is known as SQL optimisation. Reducing resource usage, shortening query execution times and enhancing overall database performance are the objectives of optimisation.

Requirement for SQL Query Optimisation

Optimising SQL queries is mostly about minimising the strain on system resources and delivering precise answers faster. It improves the efficiency of the code, which is crucial for query performance. The following are the main causes of SQL query optimisations:

- Enhancing Performance:Reducing response times and improving query speed are the primary goals of SQL Query Optimisation. For improved user experience, the time lag between a request and a response must be kept to a minimum.
- Reduced Execution Time: Faster results are obtained because of the SQL query optimisation, which guarantees less CPU time. Furthermore, it guarantees that there are no noticeable lags and that websites react rapidly.
- Enhances the Efficiency:Query optimisation cuts down on hardware maintenance time, allowing servers to operate more effectively with less power and memory usage.

Best Practices for SQL Query Optimisation

1. Use Where Clause instead of having

   The efficiency is greatly increased when the Where clause is used in place of the Having clause. Where queries execute more fast than having. Whereas having filters recorded after groups are created and when filters recorded before to group creation This indicates that utilising Where rather than having will improve efficiency and cut down on time. Read the SQL - Where Clause article to learn more about where clauses.

   For Example:

   SELECT name FROM table_name WHERE age>=18; – results in displaying only those names whose age is greater than or equal to 18 whereas

   SELECT age COUNT(A) AS Students FROM table_name GROUP BY age HAVING COUNT(A)>1; – results in first renames the row and then displaying only those values which pass the condition

2. Avoid Queries inside a Loop

   One of the greatest optimisation strategies that you should use is this one. The execution time will be significantly slowed down if queries are run inside the loop. Generally

Notes

speaking, bulk data entry and updating is possible, which is a far better strategy than using queries inside of loops.

The iterative pattern which could be apparent in loops such as for, while and do-while requires a lot of time to execute and so the performance and scalability are also affected. All of the queries can be made outside of loops to prevent this, which will increase efficiency.

3. Use Select instead of Select *

   Lowering the database load is one of the finest methods to increase efficiency. One way to achieve this is to restrict the quantity of data that is obtained from each query. Executing queries that utilise Select * will yield all pertinent data present in the database table. It will increase the database's workload by retrieving all of the superfluous data, which takes a long time, from the database.

   Let's use an example to help us better comprehend this. Take a look at a table called GeeksforGeeks that contains columns named DSA, Python and Java.

   Select * from Demotest; – Gives you the complete table as an output whereas

   Select condition from Demotest; – Gives you only the preferred(selected) value

   So the preferable method is to use a Select statement with set parameters to retrieve only necessary information. Select improves performance and lessens the strain on the database.

4. Add Explain to the Beginning of Queries

   Describe the SQL query execution process using keywords. This explanation covers many more topics as well as the order and joins of the tables. It is a handy query optimisation tool that further aids in learning the step-by-step details of execution. Provide an explanation and confirm whether or not the modifications you made resulted in a noticeable runtime reduction. Explain queries should only be run during the query optimisation process because they are time-consuming. To gain more details about Explain Queries click here.

5. Keep Wild cards at the End of Phrases

   In a string, a wildcard can be used to replace one or more characters. When using the LIKE operator, it is used. When used with a where clause, the LIKE operator looks for a given pattern. All records that match between the two wildcards will be checked when a leading wildcard and an ending wildcard are paired. Let's use an example to better grasp this.

   Examine a table called Employee, where the columns are name and salary. There are two distinct workers: Balram and Rama.

   Select name, salary From Employee Where name like '%Ram%';

   Select name, salary From Employee Where name like 'Ram%';

   In both situations, searching %Ram% will now get both Rama and Balram results, whereas searching Ram% will only yield Rama. Think about this if there are several records demonstrating how adding wild cards to the end of phrases can increase efficiency.

6. Use Exist() instead of Count()

   To find out if a particular record exists in the table or not, use the Exist() and Count() functions. However, Exist() is typically far more efficient than Count(). While Count() will continue to run and return all matching records, Exist() will continue to run until it

discovers the first matching entry. As a result, SQL query optimisation is a very time- and computational-efficient technique. While COUNT(*) continues to count rows even after passing the test, EXISTS ends when the logical test is satisfied.

Notes

Metrics for analysing query performance for SQL Query Optimisation

The cost of the query can be determined using a number of measures related to space, time, CPU usage and other resources, including:

1. Execution Time: The query execution time is the most crucial parameter for analysing query performance. The amount of time required for a query to return information from the database is known as the execution time or query duration. We may use the following commands to determine the duration of the query:

```
SET STATISTICS TIME ON
SELECT * FROM SalesLT.Customer;
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(847 rows affected)
Table 'Customer'. Scan count 1, logical reads 36, physical reads 0, page server reads 0, read-ahead reads 0,

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 151 ms.

Completion time: 2021-10-03T14:16:24.5853694+05:30
```

The query's parse, compile, execute and finish times can all be seen by using STATISTICS TIME ON.

Parse and Compile Time: Parse and Compile time is the amount of time needed to parse and compile the query in order to verify its syntax.

Execution Time: Execution time is the amount of CPU time that the query required to retrieve the data.

Completion time: Completion time is the precise moment the query returned the response.

We can clearly see whether the query is operating at a high level or not by examining these timeframes.

2. Statistics IO: while a query is made, the majority of the time is spent in IO while accessing the memory buffers for reading operations. It offers information on query execution bottlenecks and latency. We may obtain the total number of logical and physical reads needed to complete the query by turning on STATISTICS IO.

```
SET STATISTICS IO ON
SELECT * FROM SalesLT.Customer;
```

```
(847 rows affected)
Table 'Customer'. Scan count 1, logical reads 36, physical reads 0, page server reads 0,
read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0,
lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.
```

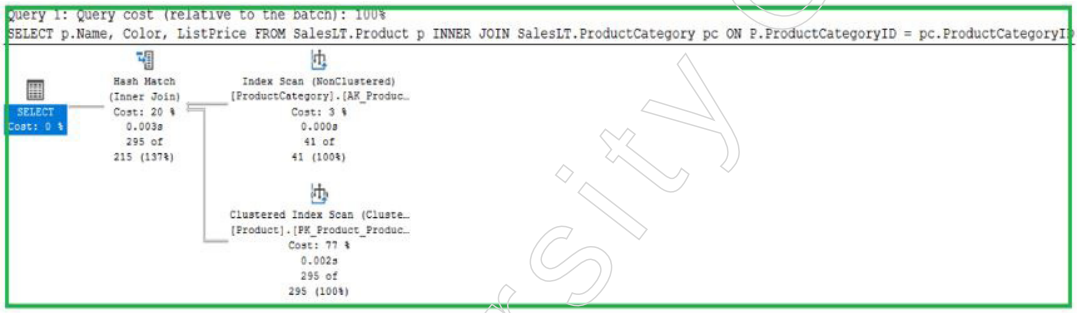
- Logical reads: The quantity of reads from the buffer cache that were completed.

Notes

- ❖ Physical reads: The quantity of reads that were made from the storage device since they weren't in the cache.

3. Execution Plan: An execution plan is a thorough, sequential processing strategy that the optimiser uses to retrieve the rows. It can be activated in the database using the following technique. It aids in the analysis of the key stages involved in running a query. Additionally, we may identify the sub-part of the execution that is consuming longer and optimise it.

```
SELECT p.Name, Color, ListPrice FROM SalesLT.Product p
INNER JOIN SalesLT.ProductCategory pc
ON P.ProductCategoryID = pc.ProductCategoryID;
```



The execution plan, as we can see above, lists the tables that were consulted and the index searches that were done to retrieve the data. It is evident how these tables were combined if joins are present.

Additionally, a more thorough analysis view of every sub-operation carried out throughout query execution is visible. Let's examine the index scan analysis:

Index Scan (NonClustered)

Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	41
Actual Number of Rows for All Executions	41
Actual Number of Batches	0
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.0033271 (3%)
Estimated CPU Cost	0.0002021
Estimated Subtree Cost	0.0033271
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	41
Estimated Number of Rows Per Execution	41
Estimated Number of Rows to be Read	41
Estimated Row Size	11 B
Actual Rebinds	0
Actual Rewinds	0

You can see that the numbers of the number of rows read, the number of actual batches, the number of executions, the estimated cost of the operator, the estimated cost

Notes

of the CPU, the estimated cost of the subtree and the actual rebinds are all available. This provides us with a thorough summary of all the expenses related to query execution.

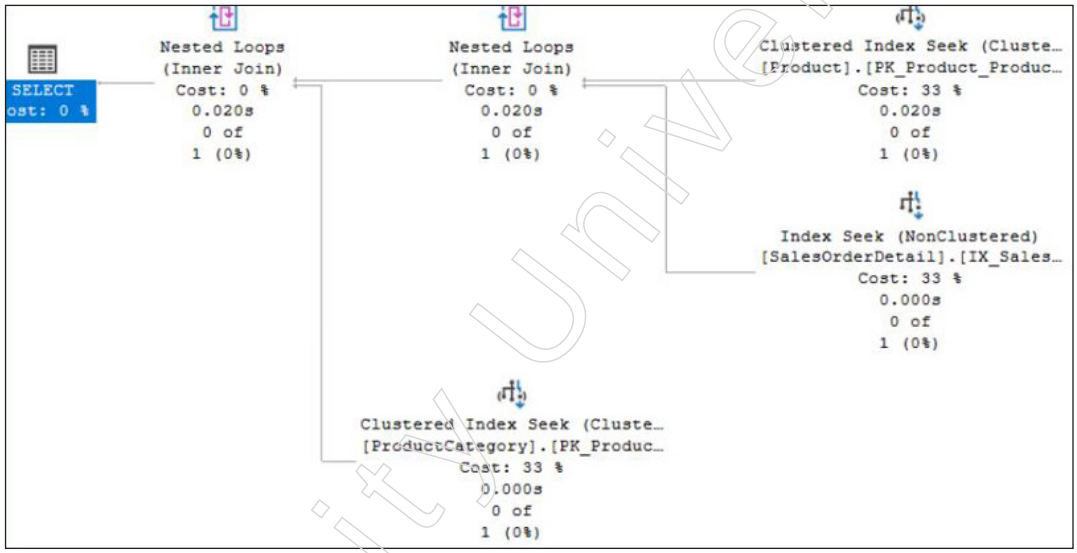
The following methods for SQL optimisation are available:

Indexing: Performance of queries can be greatly enhanced by properly indexing the tables. Indexes create data structures that speed up searching and sorting, making it possible for the database to find and get data more quickly.

A data structure called an index is utilised to give users easy access to a table using a search key. It minimises the amount of disc access required to get database rows. A seek or a scan can be used in an indexing procedure. Whereas index seek filters rows based on a matching filter, index scans search the entire index for criteria that match.

As an illustration,

```
SELECT p.Name, Color, ListPrice FROM SalesLT.Product p
INNER JOIN  SalesLT.ProductCategory pc
ON P.ProductCategoryID = pc.ProductCategoryID
INNER JOIN  SalesLT.SalesOrderDetail sod
ON p.ProductID = sod.ProductID
WHERE p.ProductID>1
```



In the above query, we can see that a total of 99% of the query execution time goes in index seek operation. Therefore, it is an important part of the optimisation process.

Guidelines for choosing index:

- Keys that regularly appear in join statements and WHERE clauses should have indexes created for them.
- Columns that are often modified—that is, when the UPDATE command is executed frequently—should not have indexes created on them.
- Foreign keys where INSERT, UPDATE and DELETE are executed concurrently should have indexes created for them. This prevents shared locking on the weak entity and permits UPDATES on the master table.
- Using the AND operator, indexes should be created on properties that frequently occur together in the WHERE clause.

Notes

- Indexes based on key value ordering ought to be created.

Selection

It is recommended to choose the necessary rows rather than all of the rows. Because SELECT * searches the whole database, it is incredibly wasteful.

```
SET STATISTICS TIME ON
SELECT * FROM SalesLT.Product
```

```
(295 rows affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 125 ms.

Completion time: 2021-10-03T17:45:01.8602635+05:30
```

```
SET STATISTICS TIME ON
SELECT ProductNumber, Name, Color,Weight FROM SalesLT.Product
```

```
(295 rows affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 41 ms.

Completion time: 2021-10-03T17:45:58.3009888+05:30
```

The two outputs above demonstrate how using the SELECT command to select only the necessary data cuts down on time to a quarter.

Avoid using SELECT DISTINCT

In SQL, the SELECT DISTINCT command is used to extract unique results and eliminate duplicate rows from a relation. It essentially puts together relevant rows and then removes them in order to accomplish this objective. The GROUP BY procedure is an expensive one. Therefore, additional attributes may be used in the SELECT process to retrieve unique rows and eliminate duplicate rows.

Let's use an illustration here.

```
SET STATISTICS TIME ON
SELECT DISTINCT Name, Color, StandardCost, Weight FROM SalesLT.Product
```

```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 111 ms.

Completion time: 2021-10-03T23:13:53.9723237+05:30
```

```
SET STATISTICS TIME ON
SELECT  Name, Color, StandardCost, Weight, SellEndDate, SellEndDate FROM SalesLT.Product
```

Notes

```
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 69 ms.

Completion time: 2021-10-03T23:13:19.7751003+05:30
```

The execution of the two queries above shows that it takes longer for the DISTINCT operation to retrieve the unique rows. Therefore, to increase performance and obtain unique rows, it is preferable to include more attributes in the SELECT query.

Inner joins vs WHERE clause

When combining two or more tables, we should utilise inner join instead of the WHERE clause. The CROSS join/CARTESIAN product is created for table merging by the WHERE clause. The CARTESIAN product of two tables is time-consuming.

```
SET STATISTICS IO ON
SELECT p.Name, Color, ListPrice
FROM SalesLT.Product p, SalesLT.ProductCategory pc
WHERE P.ProductCategoryID = pc.ProductCategoryID
```

```
(295 rows affected)
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads 0,
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads 0
Table 'Product'. Scan count 1, logical reads 103, physical reads 0, page server reads 0, read-ahead reads 0
Table 'ProductCategory'. Scan count 1, logical reads 2, physical reads 0, page server reads 0, read-ahead r

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 90 ms.

Completion time: 2021-10-03T23:33:29.2651153+05:30
```

```
SET STATISTICS TIME ON
SELECT p.Name, Color, ListPrice FROM SalesLT.Product p
INNER JOIN  SalesLT.ProductCategory pc
ON P.ProductCategoryID = pc.ProductCategoryID
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 21 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(295 rows affected)
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads 0,
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads 0,
Table 'Product'. Scan count 1, logical reads 103, physical reads 0, page server reads 0, read-ahead reads 0,
Table 'ProductCategory'. Scan count 1, logical reads 2, physical reads 0, page server reads 0, read-ahead re
```

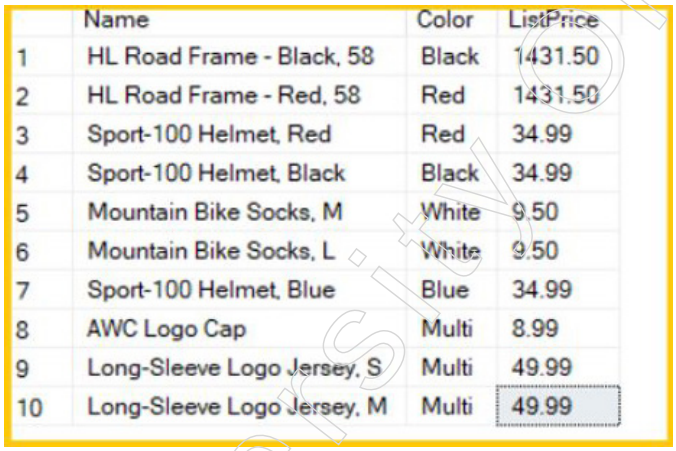
The aforementioned outputs demonstrate that, in comparison to joins made with a WHERE clause, inner joins complete roughly half as quickly.

LIMIT command

Notes

To regulate how many rows from the result set are shown, use the limit command. Only the necessary rows must be displayed in the result set. Consequently, limits must be applied to the production dataset and an on-demand row computing feature must be provided for production purposes.

```
SET STATISTICS IO ON
SELECT Name, Color, ListPrice
FROM SalesLT.Product
LIMIT 10
```



	Name	Color	ListPrice
1	HL Road Frame - Black, 58	Black	1431.50
2	HL Road Frame - Red, 58	Red	1431.50
3	Sport-100 Helmet, Red	Red	34.99
4	Sport-100 Helmet, Black	Black	34.99
5	Mountain Bike Socks, M	White	9.50
6	Mountain Bike Socks, L	White	9.50
7	Sport-100 Helmet, Blue	Blue	34.99
8	AWC Logo Cap	Multi	8.99
9	Long-Sleeve Logo Jersey, S	Multi	49.99
10	Long-Sleeve Logo Jersey, M	Multi	49.99

The resultset's top 10 rows are printed by the query mentioned above. This significantly enhances the query's performance.

IN versus EXISTS

When it comes to scan costs, the IN operator is more expensive than the EXISTS, particularly when the subquery yields a huge dataset. Therefore, when retrieving results with a subquery, we ought to attempt to use EXISTS rather than IN.

Let's examine this with an example.

```
SET STATISTICS IO ON
SELECT Name, Color, ListPrice
FROM SalesLT.Product
LIMIT 10
```

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 2 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(295 rows affected)
Table 'ProductDescription'. Scan count 1, logical reads 590
Table 'Product'. Scan count 1, logical reads 103, physical

 SQL Server Execution Times:
   CPU time = 16 ms,  elapsed time = 111 ms.
```

Notes

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 4 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

(295 rows affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 53 ms.

Completion time: 2021-10-04T17:41:52.7784161+05:30
```

The same query with a subquery including both IN and EXISTS instructions was run and we noticed that the EXISTS command takes half the time compared to the IN command and that there are very few logical and physical scans.

Loops versus Bulk insert/update

Since it necessitates repeatedly repeating the same query, loops must be avoided. We ought to choose bulk inserts and updates instead.

```
SET STATISTICS TIME ON
DECLARE @Counter INT
SET @Counter=1
WHILE ( @Counter <= 10)
BEGIN
PRINT 'The counter value is = ' + CONVERT(VARCHAR,@Counter)
INSERT INTO [SalesLT].[ProductDescription]
([Description]
,[rowguid]
,[ModifiedDate])
VALUES
('This is great'
,NEWID()
,'12/01/2010')
SET @Counter  = @Counter  + 1
END
```

```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 7 ms.

(9 rows affected)

Completion time: 2021-10-04T18:16:06.5049276+05:30
```

Notes

```
USE [AdventureWorksLT2019]
GO
SET STATISTICS TIME ON
INSERT INTO [SalesLT].[ProductDescription]
([Description]
,[rowguid]
,[ModifiedDate])
VALUES
('This is great'
,NEWID()
'12/01/2010'),
('New news'
,NEWID()
,'12/01/2010'),
('Awesome product.'
,NEWID()
,'12/01/2010'),
.....,
('Awesome product.'
,NEWID()
,'12/01/2010')
GO
```

As demonstrated above, bulk inserts operate more quickly than loop expressions.

1.1.6 Evaluation

Let us first go over how a SQL query can be represented before moving on to the evaluation of a query expression. Examine the following relationships between students and marks:

STUDENT (enrolno, name, phone)

MARKS (enrolno, subjectcode, grade)

The following inquiry may be used to obtain the results of the student or students whose phone number is "1129250025."

```
SELECT enrolno, name, subjectcode, grade
FROM STUDENT s, MARKS m
WEHRE s.enrolno=m.enrolno AND phone= '1129250025'
```

The equivalent relational algebraic query for this will be:

$\pi^{enrolno,name,subjectcode}(\sigma^{phone='1129250025'}(STUDENT)\ MARKS)$

This is a very good internal representation, but for ease of designing query optimisation algorithms, it might be a good idea to represent the relational algebraic expression to a

query tree, where relations represent the leaf and nodes are the operators. The query tree for the relational expression above would be:

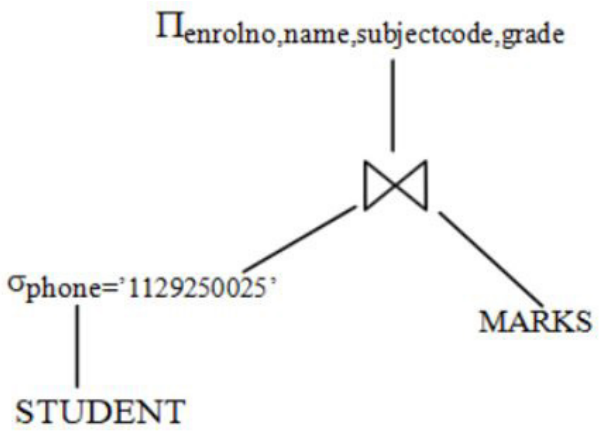


Figure: A Sample query tree

We have seen the algorithms for specific operations in the previous section. Let's now examine the techniques for assessing an expression as a whole. Generally, we employ two techniques:

- Materialisation
- Pipelining

Materialisation: From the bottom up, evaluate a relational algebraic statement by producing and storing the output of each operation explicitly. For instance, in the aforementioned Figure, the selection operation result on the STUDENT relation is computed and stored, followed by a join of this result with the MARKS relation and the compilation of the projection operation. Although it is always possible to perform a materialised examination, writing and reading results to and from disc might be quite expensive.

Pipelining: Evaluate operations while the first is running in a multi-threaded fashion, that is, by passing the tuples output of one operation as input to the next parent operation. Instead of storing (materialising) outcomes, the prior expression tree sends tuples straight to the join. Likewise, sends tuples straight to the projection and does not keep the results of the join. Therefore, it is not necessary for each operation to save a temporary relation on a disc. For sort and hash-join, pipelining might not always be feasible or simple.

A buffer filled with the result tuples of a lower level operation may be used in one of the pipelining execution techniques and the higher level operation may retrieve records from the buffer.

Creation of Query Evaluation Plans

Can anything be done in these two steps to optimise the query evaluation, as we have previously covered query representation and its ultimate evaluation in the previous section? This section goes over this procedure in great depth.

There are multiple steps involved in creating query-evaluation plans for an expression:

- Using equivalency rules to create logically equivalent expressions
- Annotating generated expressions to obtain different query strategies
- selecting the least expensive option based on projected expenses.

We refer to the entire procedure as cost-based optimisation.

There would be a huge cost differential between a good and a bad query evaluation

Notes

mechanism. As a result, we would need to calculate the operational costs and gather relationship statistics. For instance, several tuples, several unique attribute values, etc. Calculating the cost of complex expressions can be made easier by using statistics to estimate intermediate results.

Transformation of Relational Expressions

If two relational algebraic expressions produce the same set of tuples on each instance of a legal database, they are said to be equivalent (tuple order is not important).

Let's describe some equivalency criteria that can be applied to produce relational expressions that are equivalent.

Equivalence Rules

1. A series of separate selections can be compared to the conjunctive selection operations. It is represented by:

   $s_{q1Ùq2}(E) = \sigma_{\theta 1} (\sigma_{\theta 2} (E))$

2. Because the selection processes are commutative,

   $s_{q1}(s_{q2}(E) = \sigma_{\theta 2} (\sigma_{\theta 1} (E))$

3. The final projection operation in the sequence is required; the remaining ones can be skipped.

   $\pi_{attriblist1} (\pi_{attriblist2} (\pi_{attriblist3}\text{.....}(E) ..) = \pi_{attriblist1} (E)$

4. Cartesian products and theta join operations can be mixed with the selection procedures.

   $\sigma_{\theta 1} (E1 \times E2) = E1 \bowtie_{\theta 1} E2$

   and

   $\sigma_{\theta 2} (E1 \bowtie_{\theta 1} E2) = E1 \bowtie_{\theta 2 \wedge \theta 1} E2$

5. The theta-join procedures and natural joins are commutative.

   $E1 \bowtie_{\theta} E2 = E2 \bowtie_{\theta} E1$

6. Associative join procedures are provided by Natural. Theta joins have the appropriate distribution of joining conditions, but they are likewise associative:

   $(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$

7. When all of the characteristics in the selection predicate involve just the attributes of one of the expressions (E1) being joined, then the selection operation distributes throughout the theta join operation.

   $\sigma_{\theta 1}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta 1}(E1)) \bowtie_{\theta} E2)$

8. Only those attributes that are present in that relation are distributed over the theta join operation by the projections operation.

   $\pi_{attrlist1} \bowtie_{attrlist2} (E1 \bowtie_{\theta} E2) = (\pi_{attrlist1} (E1) \bowtie_{\theta} \pi_{attrlist2} (E2))$

9. Union and intersection are commutative set operations. Set difference, however, does not commute.

   $E1 \cup E2 = E2 \cup E$ and similarly for the intersection.

10. Additionally associated are set union and intersection operations.

    $(E1 \cup E2) \cup E3 = E1 \cup (E2 \cup E3)$ and similarly for intersection.

11. The union, intersection and setdifferences operations can all be used to spread the selection operation.

Notes

$\sigma_{\theta 1}$ (E1 – E2) = ((σ θ1 (E1) - ($\sigma_{\theta 1}$ (E2))

12. One can distribute the projection operation within the union.

$\pi_{attriblist1}$ (E1 ∪ E2) = $\pi_{attriblist1}$ (E1) ∪ $\pi_{attriblist1}$(E2)

The aforementioned principles are overly broad and they may yield a few heuristic rules that aid in improving the relational expression. These guidelines are:

1. Testing each predicat on the tuples simultaneously while combining a series of selections into a conjunction:

   $\sigma_{\theta 2}$ ($\sigma_{\theta 1}$ (E)) convert to $\sigma_{\theta 2 \wedge \theta 1}$ (E)

2. Creating a single outer projection from a cascade of projections:

   π_4(π_3 (……(E))) = π_4 (E)

3. In certain cases, cost can be decreased by switching the projection and selection.
4. For the Cartesian product or joining, apply associative or commutative principles to identify different options.
5. Moving the selection and projection (it may have to be adjusted) before joining. Because fewer tuples are created as a result of the projection and selection, the joining cost could be decreased.
6. Commuting the projection and selection with Cartesian product or union.

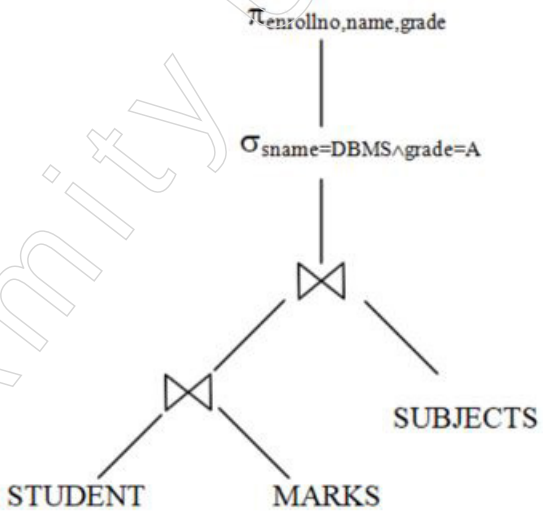
Let's use an example to clarify how to apply a few of these criteria. Think on this question for the relations:

STUDENT (enrollno, name, phone)

MARKS (enrollno, subjectcode, grade)

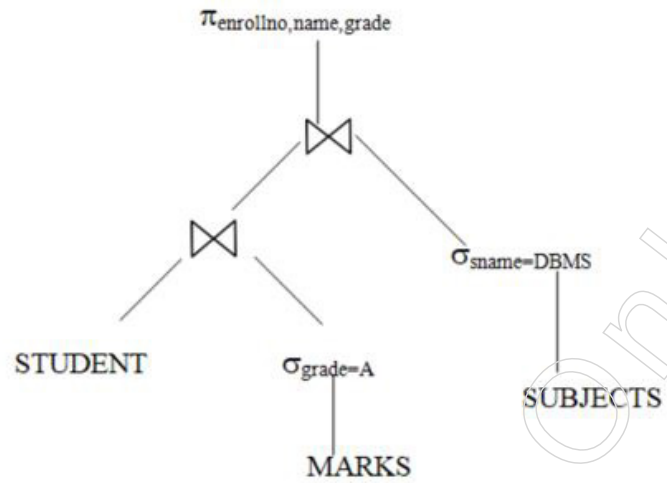
SUBJECTS (subjectcode, sname)

Consider the query: Locate the student's name, grade and enrollment number for individuals who have earned an A in the DBMS course. Here are a few potential answers to this question:

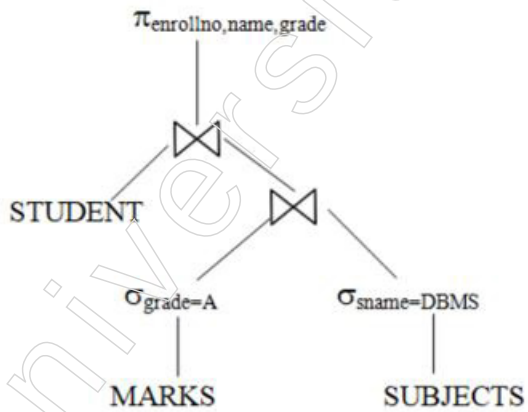


It is possible to shift the selection condition to the join operation. The preceding figure specifies the selection criteria as follows: sname = DBMS and grade = A. Since the grade is in the MARKS table and the name is only available in the SUBJECTS table, these two requirements pertain to distinct tables. As a result, the mapping of the selection criteria will be done as the figure below illustrates. As a result, the comparable phrase will be:

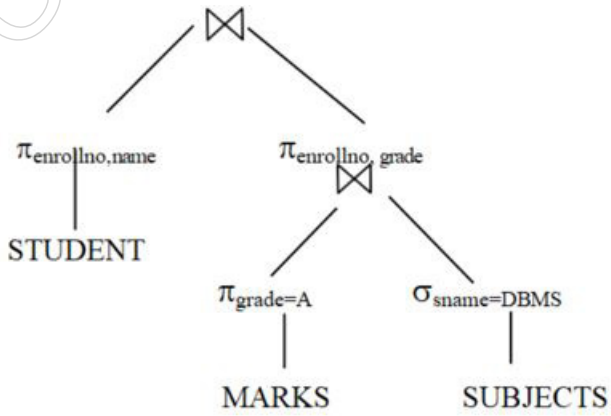
Notes



It could be a good idea to first join MARKS with SUBJECTS because the projected size of SUBJECTS and MARKS after selection will be small. Thus, it is possible to apply the associative law of JOIN.



Lastly, the projection can be shifted within. Hence, the query tree that results could be:



Obtaining Alternative Query Expressions

Equivalency rules are used by query optimisers to systematically produce expressions that are equivalent to the input expression. From a conceptual standpoint, they produce every equivalent expression by iteratively applying the equivalency criteria until no more expressions can be produced. Add newly created expressions to the list of expressions already identified for each expression that has been found, making use of all available equivalency rules. Unfortunately, the aforementioned method is highly costly in terms of

both time and area. The aforementioned heuristic guidelines can be applied to cut expenses and generate a few viable, high-quality query expressions.

Notes

Query Evaluation Plans

An assessment plan outlines the precise method to be utilised for each operation as well as the coordination of its execution. Figure 6 displays the query tree along with the assessment plan, for instance.

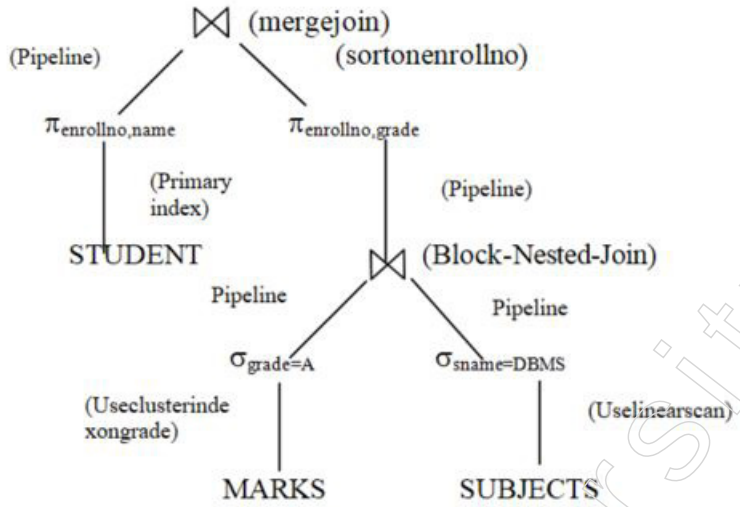


Figure: Query evaluation plan

Choice of Evaluation Plans

The interplay of evaluation methodologies must be taken into account while selecting one. Please be aware that selecting the least expensive method for every action on its own might not result in the optimal algorithm overall. Merge-join, for instance, could be more expensive than hash-join, but it might offer a sorted result, which lowers the cost for a higher level aggregate. A nested-loop connect could present a pipelining opportunity. Realistic query optimisers combine components of the two main strategies listed below:

- Searches all the plans and chooses the best plan in a cost-based fashion.
- Uses heuristic rules to choose a plan.

1.1.7 Query Tree

A query tree is a hierarchical diagram that shows the execution plan and structure of a query. It offers a graphic representation of all the different steps and connections needed to complete a database query. Usually, the parsing and optimisation stages of query processing involve building the query tree. The general process of creating a query tree is as follows:

Parsing:

First, the user's query is parsed to make sure it follows the rules of the query language and is syntactically valid. A parse tree, which depicts the query's hierarchical structure based on the query language's grammar, is created when the query is parsed.

Query Optimisation:

After that, optimisation techniques are used to the parse tree in order to increase query execution performance.

Notes

The optimiser looks at many execution strategies and determines which is the most effective plan based on variables like access techniques, join strategies and indexes that are accessible.

In most cases, a query execution tree is used to illustrate the optimised strategy.

Query Execution Tree:

The query plan is more accurately represented by the query execution tree. The steps required to retrieve and process the data are outlined in it.

Operations like projections, joins, access techniques and selects are represented by nodes in the tree.

The tree structure reflects the logical and physical flow of operations in the execution plan.

Physical Execution Plan:

A physical execution plan that details how the database engine will carry out the activities is created by further transforming the query execution tree.

This plan considers various elements, including access methods, available indexes and data distribution.

Execution:

The execution engine receives the final physical execution plan, interprets it and arranges for the database's contents to be retrieved and processed.

The coordinated execution of the operations is ensured by the execution engine adhering to the query tree's structure.

Steps to Make a Query Tree

Step 1:

To obtain the resulting tuples that we use for the next operation, execute the leaf nodes with their associated internal nodes that have the relational algebra operator with the given conditions.

Step 2:

This operation is repeated until we reach the root node, at which point, using the specified conditions, we PROJECT (π) the necessary tuples as the output.

Let's use a few instances to better grasp this:

Example 1:

Examine this expression in relational algebra:

πp (R ⋈R.P = S.P S)

Step 1: As the Leaf nodes of the tree, write the relations you wish to implement. Here, the relations are R and S.

Figure:Two Relations R and S

Notes

Step 2: As an internal node (or parent node of these two leaf nodes), add the condition (in this case, R.P = S.P) with the relational algebra operator.

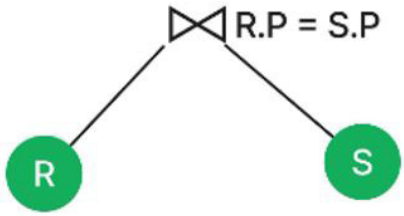


Figure: JOIN R and S where R.P = S.P

Step 3: Add the root node now, which produces the query's output when it executes.

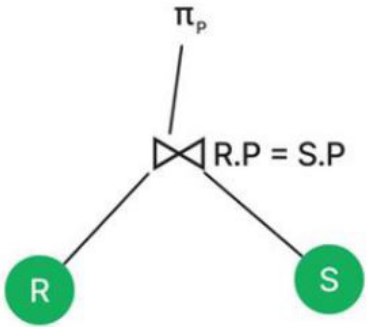


Figure: Execution Output

Database administrators and developers can better understand and optimise the execution of complex queries by using the query tree. To help with performance optimisation and debugging, it offers a visual depiction of the logical and physical steps involved in processing a query.

1.1.8 Query Graph

A query graph is a visual depiction of the steps or actions a database management system (DBMS) would carry out in order to execute a particular query. It is also referred to as a query execution plan or query plan. The data flow and linkages between various actions involved in executing a database query are depicted in the query graph. During the optimisation stage of query processing, the query optimiser creates query graphs.

Role of Query Graph in SQL

The optimisation and execution of SQL queries within a database management system (DBMS) are strongly related to the function of a query graph. In the context of SQL, let's examine the primary responsibilities and purposes of a query graph:

Query Optimisation:

The DBMS must decide how to execute a SQL query in the most effective manner when a user submits one. The process of query optimisation entails creating and assessing several execution strategies.

The selected execution plan is represented visually via the query graph. It describes the steps that the DBMS will take in order to answer the query.

Visual Representation:

The mental and physical processes involved in running a SQL query are represented visually by the query graph.

Notes

SQL procedures including table scans, index scans, joins, filters, projections and aggregations are represented as nodes in the graph. Edges show how data moves between several operations.

Query Execution Plan:

The plan for executing the query is fundamentally represented by the query graph. The plan describes how the DBMS will access and handle the necessary data.

Understanding the query execution plan is essential to comprehending how the DBMS will access the database to obtain the needed data.

Cost-Based Optimisation:

The query optimiser assesses several query execution techniques and allocates costs to each plan according to variables including join strategies, available indexes and the anticipated number of rows.

The query graph is a tool for comprehending and visualising the selected execution plan and the objective is to choose the execution plan with the lowest cost.

Parallel Execution:

In order to enhance query performance, several contemporary DBMSs allow for the concurrent execution of specific tasks.

There may be parallel operations in the query graph, indicating portions of the query that can run simultaneously.

Debugging and Performance Tuning:

The query graph is used by developers and database administrators for performance optimisation and debugging. They can locate possible bottlenecks, ineffective processes and places for optimisation by examining the graph.

Query Explain Plans:

DBMSs frequently come with tools for creating and viewing explanation plans, which are visual or textual depictions of the query graph.

Explain plans give users insight into optimisation choices and explain how the DBMS will process their queries.

1.1.9 Heuristic Optimisation of Query Tree

Many relational database management systems have a feature called query optimisation, which looks at several query plans to satisfy a query and determines which query plan is the best. Since there are various approaches to creating plans, this may or may not be the best course of action.

The amount of time required to determine the optimal strategy and the quantity required to implement it are subject to trade-offs. Different database management system attributes have different strategies for striking a balance between these two. Query Optimisers are one of the key mechanisms by which current database systems gain their performance advantages.

An optimiser will select the best strategy from the numerous available options for assessing a request for data retrieval. Since query optimisers are already among the biggest and most intricate database system modules, it has been challenging to expand and change them to fit these uses.

Notes

Within the database field, query optimisation covers a wide range of topics. It has been examined from a wide range of perspectives and in a wide range of circumstances, leading to a number of unique answers in each instance. SQL has been the de facto standard for relational query languages over time. The query optimiser and the query execution engine are two essential parts of a SQL database system's query evaluation component.

This original query tree will be converted into an equivalent final query tree that can be executed efficiently by the heuristic query optimiser.

Rules for equivalency between relational algebra expressions that can be used to convert the original tree into the final, optimised query tree must be included in the optimiser. We first go over the informal transformation of a query tree using heuristics and then we go over generic transformation rules and demonstrate their use in an algebraic heuristic optimiser.

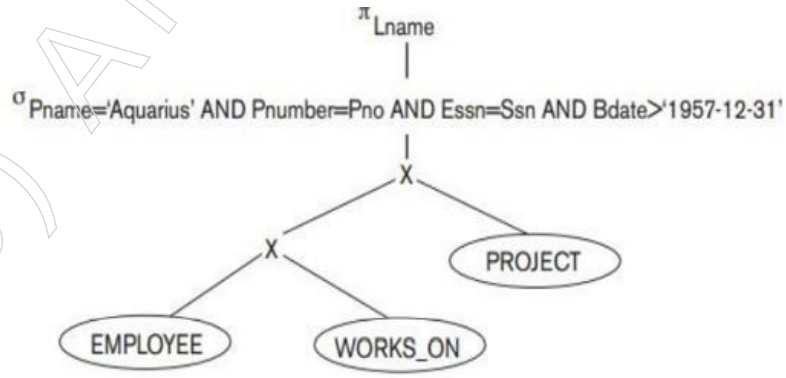
1. Create a cascade of select operations by dividing SELECT operations with conjunctive conditions.
2. Move each SELECT operation as far down the query tree as is allowed by the characteristics included in the select condition, making use of the commutativity of SELECT with other operations.
3. Rearrange the tree's leaf nodes by using the binary operations' commutativity and associativity.
4. If the condition is a join condition, then combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree to create a JOIN operation.
5. Lists of projection attributes should be broken down and moved as far down the tree as possible using the cascading and commuting properties of PROJECT and other operations. New PROJECT operations should be created as needed.
6. Identify sub-trees that reflect groups of operations that can be executed by a single algorithm.

Request "Find the last names of employees born after 1957 who work on a project named ‗Aquarius'."

SQL

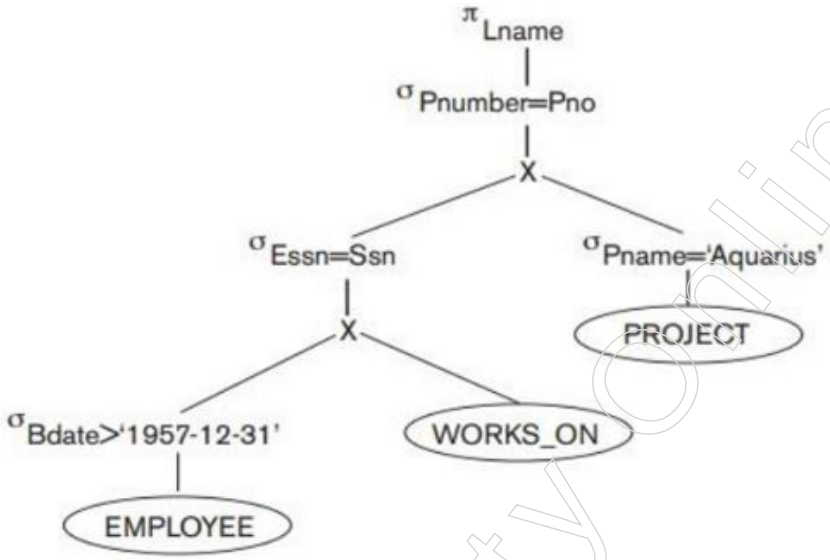
```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE PNAME=‗Aquarius'AND PNUMBER=PNO AND ESSN=SSN AND BDATE.‗1957-12-31';
```

a.

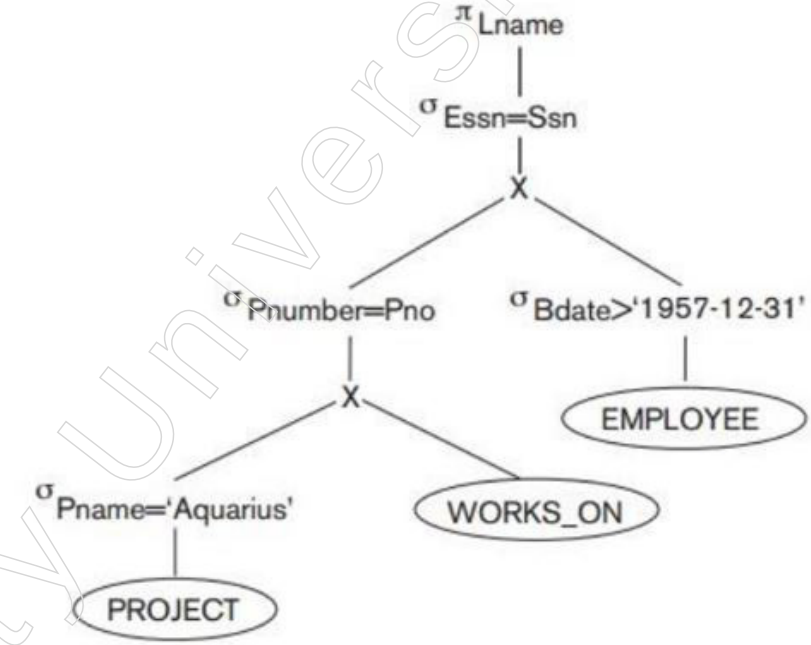


Notes

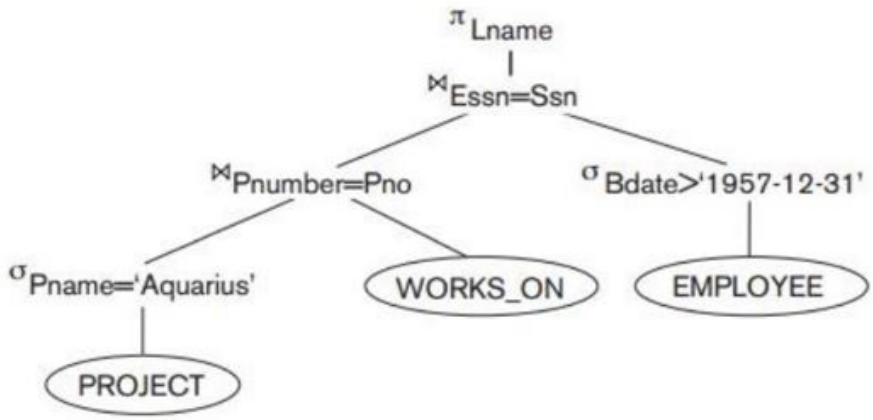
b.



c.

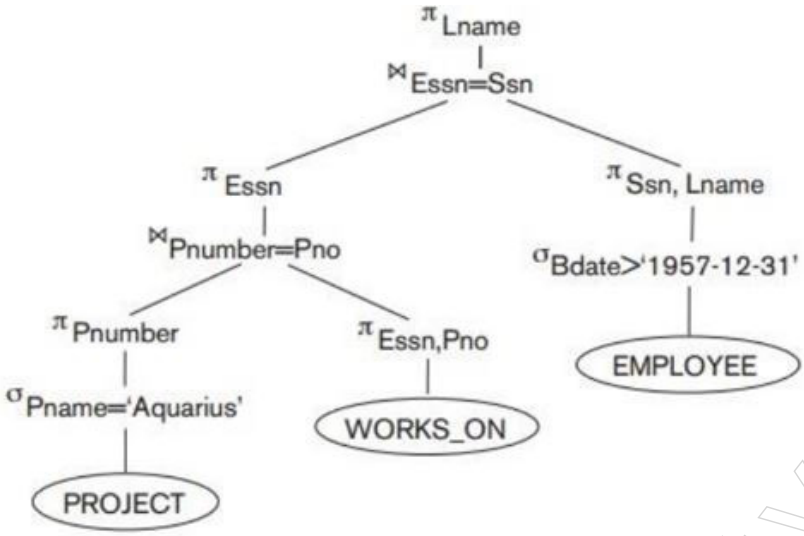


d.



Notes

e.



Different Query Optimisation Approaches

Cost Based Optimisation

This is how a cost-based query optimiser operates: It first creates every query execution plan that could be possible. Next, an estimate of each plan's cost is provided. Lastly, the plan with the lowest expected cost is selected based on the estimation. The chosen course of action might not be the best one because the selection is based on predicted cost values.

The intricacy and precision of the cost functions employed determine the calibre of optimiser choices. It uses a variety of methods, such as dynamic programming, to determine the optimal strategy. The primary disadvantage of it is its high cost. Because of this, very few optimisers use this tactic. A cost estimating technique is so that a cost may be allocated to each plan in the search space. This seems to be an estimate of the resources required to carry out the plan.

1. Generates all possible query execution plans and then cost is Calculate
2. Quality depends on complexity and accuracy of cost Function.

Cost-based query Optimisation:

Algebraic Formulas for the Given Question:

SELECT p.pname, d.dname FROM Patients p, Doctors d WHERE p.doctor = d.dname AND d.dgender ='M'

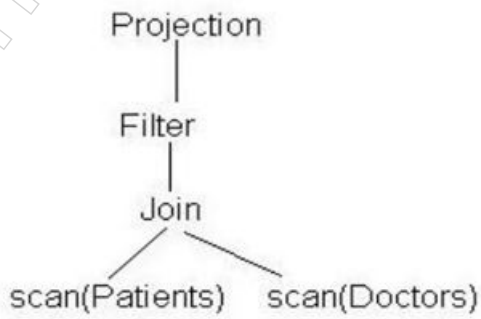


Figure: Relational Algebra Expression for Query

Notes

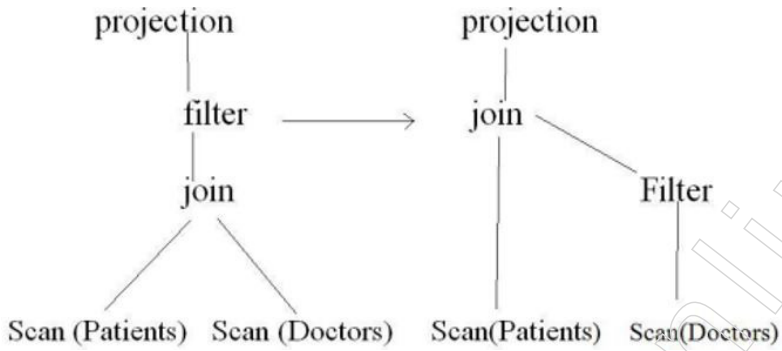


Figure: Execution Plan

Advantages:

- Instead of taking time constraints into account, adjusts to client needs
- The speed at which queries are retrieved has increased.

Disadvantages:

- utilises cost-based optimisation, making it pricey.

Semantic Query Optimisation

If two queries yield the same result for a database, they are considered semantically similar. Integrity constraints are used for this reason in order to match outcomes. Finding the set of semantic changes that produce a semantically comparable question at a lower execution cost is known as semantic query optimisation.

Using the Integrity Constraint Rules, ODB Optimiser decides which more specialised classes to visit and minimises the number of components.

Advantages

1. accepts recursive questions as well as disjunction and negation queries.

Disadvantages:

1. Only appropriate for basic prototypes.
2. There are no commercial versions available.

Rules

Heuristic optimisation uses a set of rules to increase performance by transforming the expression-tree. These guidelines are listed below. −

- Start the query's SELECTION procedure first. For each SQL table, this ought to be the initial step. Instead of using all the tables in the query, we can accomplish this by reducing the number of records needed.
- Complete every projection in the query as soon as it is possible. Similar to a selection, but with fewer columns in the query thanks to this technique.
- Execute the joins and selection procedures that are the most restricted. This means that you should limit your selection to those sets of tables and/or views that are absolutely required for the query and will produce a comparatively less number of records. Naturally, joining tables with minimal records improves query performance.

Steps in heuristic optimisation

The following explains the steps involved in heuristic optimisation.

Notes

- Break down the conjunctive choices into a series of separate selection processes.
- Move the selection operations down the query tree for the earliest feasible execution.
- Perform the join operations and those selections first, as this will result in the smallest relations.
- Join operations should be used in place of the cartesian product operation and selection operation.
- Destructive: Take down the tree as much as you can.
- Determine which subtrees have pipelined operations.

1.1.10 Functional Dependencies

A relation between two qualities has a set of restrictions known as functional dependency (FD). According to functional dependency, two tuples that share the same values for attributes A1, A2,..., An also need to share the same values for attributes B1, B2,..., Bn.

If two tuples with the same value for property An also have the same value for attribute B, then there is a functional dependency A->B in the relation. A functional dependency is shown by an arrow with the symbol "→". X→ Y stands for the functional dependence of X on Y.

The examples that follow should help you grasp functional dependency:

STUDENT

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajsthan	India	18
4	SURESH		Punjab	India	21

Table 1

STUD_NO->STUD_NAME, STUD_NO->STUD_PHONE hold

but

STUD_NAME->STUD_ADDR do not hold

A connection between two qualities is called a functional dependency. It often sits in a table between the primary key and a non-key property.

X → Y

The right side of production is known as a dependant, while the left side of FD is known as a determinant.

For Eg:

Assume we have an EMP table with attributes: Emp_Id, E_NAME, Emp_Address.

Because we can determine the EMP name associated with an Emp_Id by knowing the Emp_Id, the E_NAME field of the EMP table can be uniquely identified in this case.

Functional dependence is expressed as follows:

Emp_Id → E_NAME

We can state that Emp_Id is operationally necessary for E_NAME.

We have a <Department> table with two attributes – DeptId and DeptName.

Notes

DeptId = Department ID

DeptName = Department Name

The DeptId serves as our main key. In this case, DeptName is uniquely identified by DeptId. This is due to the fact that you first need the DeptId if you want to know the department name.

DeptId	DeptName
001	Finance
002	Marketing
003	HR

As a result, it may be said that DeptId and DeptName are functionally reliant on one another in the example above.

DeptId ->DeptName

Armstrong's Axioms

Armstrong's Axioms are a set of rules that, when repeatedly applied, result in a closure of functional dependencies. If F is a set of functional dependencies, then the closure of F, denoted as F+, is the set of all functional dependencies logically inferred by F.

- Reflexive rule – If beta is_subset_of alpha and alpha is a set of qualities, then alpha holds beta.
- Augmentation rule – Ay →by additionally holds if a →b holds and y is attribute set. This changes nothing about the fundamental dependencies; just characteristics are added.
- Transitivity rule – Similar to the transitive rule in algebra, if both a→b and b→ c are true, then a→c is likewise true. Functionally speaking, a determines b and b determines a.

How to find functional dependencies for a relation?

A relation's functional dependencies depend on the relation's domain. Take into account the STUDENT connection shown in Table below.

We know that STUD_NO is unique for each student. So STUD_NO->STUD_NAME, STUD_NO->STUD_PHONE, STUD_NO->STUD_STATE, STUD_NO->STUD_COUNTRY and STUD_NO -> STUD_AGE all will be true.

- Similarly, STUD_STATE->STUD_COUNTRY will be true as if two records have same STUD_STATE, they will have same STUD_COUNTRY as well.

For relation STUDENT_COURSE, COURSE_NO->COURSE_NAME will be true as two records with same COURSE_NO will have same COURSE_NAME.

Rules of Functional Dependency

The three most crucial guidelines for functional dependencies in databases are listed below:

- Reflexive rule –. If Y is_subset_of X and X is a set of attributes, then X has a value for Y.
- Augmentation rule: Ac ->bc also holds when x -> y holds and c is attribute set. That is the addition of properties without altering the fundamental dependencies.
- Transitivity rule: This statement is extremely similar to the transitive rule in algebra, which states that if x -> y and y -> z are true, then x -> z is true as well. As a functional relationship, X determines Y.

Notes

Types of Functional Dependencies

The basic categories of functional dependencies in DBMS are four. The types of functional dependencies in DBMSs are as follows:

- Multivalued Dependency
- Trivial Functional Dependency
- Non-Trivial Functional Dependency
- Transitive Dependency

Multivalued Dependency in DBMS

When there are numerous independent multivalued characteristics in a single table, multivalued dependency occurs. A complete constraint between two sets of characteristics in a relation is a multivalued dependency. It necessitates that a relation contain specific tuples. For clarification, consider the following Multivalued Dependence Example.

Example:

Car_model	Maf_year	Color
H001	2017	Metallic
H001	2017	Green
H005	2018	Metallic
H005	2018	Blue
H010	2015	Metallic
H033	2012	Gray

In this illustration, maf_year and colour are unrelated to one another but rely on car model. These two columns in this illustration are referred to be multivalue dependant on car model.

This reliance can be illustrated as follows:

car_model ->maf_year

car_model-> colour

Trivial Functional Dependency in DBMS

A set of attributes is said to be trivial if they are included in another attribute and this is what a trivial dependency is.

Hence, if Y is a subset of X, the functional dependency X->Y is straightforward. Let's explain with the help of a simple functional dependency example.

For instance:

Emp_id	Emp_name
AS555	Harry
AS811	George
AS999	Kevin

Notes

Have a look at the table below, which has the columns Emp_id and Emp_name.

{Emp_id, Emp_name} ->Emp_id is a trivial functional dependency as Emp_id is a subset of {Emp_id,Emp_name}.

Non Trivial Functional Dependency in DBMS

When A->B holds true and B is not a subset of A, a functional dependency, also known as a nontrivial dependency, occurs. If attribute B in a connection isn't a subset of attribute A, it's referred to as a non-trivial dependency.

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Apple	Tim Cook	57

Example:

(Company} -> {CEO} (if we know the Company, we knows the CEO name)

CEO, however, is not a subset of Company, making their functional dependence non-trivial.

Transitive Dependency in DBMS

When "t" is indirectly produced by two functional dependencies, it results in a particular kind of functional dependency known as a transitive dependency. Let's clarify with the help of the subsequent transitive dependency example.

Example:

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Alibaba	Jack Ma	54

{Company} -> {CEO} (if we know the company, we know its CEO's name)

{CEO } -> {Age} If we know the CEO, we know the Age

Thus, in accordance with the transitive dependency principle:

{ Company} -> {Age} should hold, that makes sense because if we know the company name, we can know his age.

1.1.11 Normal Forms

In order to avoid data redundancy, insertion, update and deletion anomalies, normalisation is a process for organising the data in the database. It is a method for examining relational schemas based on their many beneficial dependencies and primary keys. The relational database theory is innately suited to normalisation. It may duplicate comparable data already present in the database, resulting in the creation of additional tables. If a database configuration isn't ideal, it may contain anomalies, which are every database administrator's worst nightmare. It's nearly impossible to imagine dealing with a database that has anomalies.

Notes

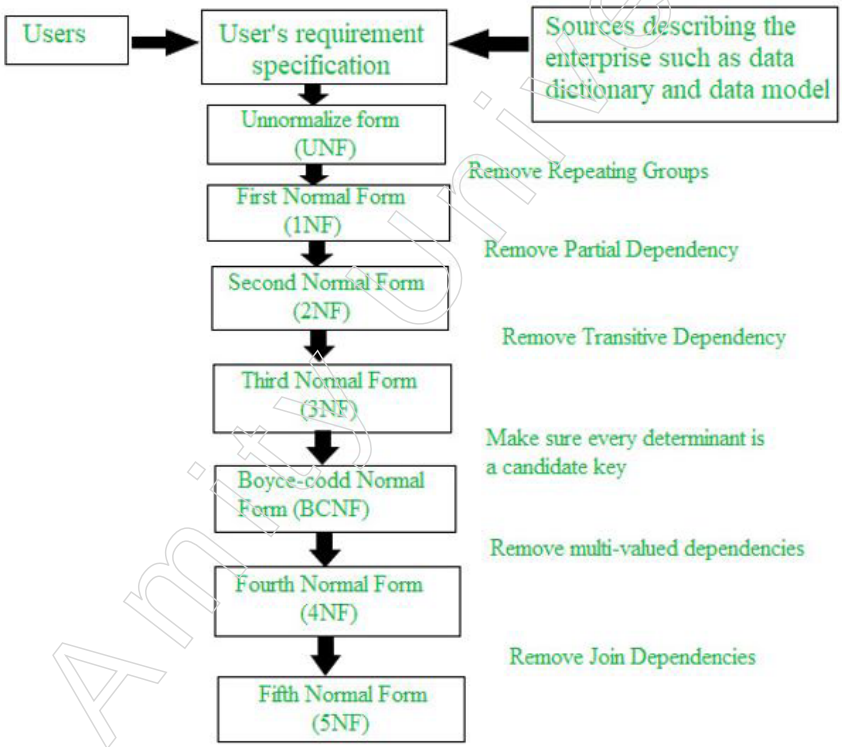
Normalisation

- The process of structuring the data in the database is called normalisation.
- Redundancy from a relation or collection of relations is reduced through the use of normalisation. Also, it is employed to get rid of unwanted traits including Insertion, Updating and Deletion Anomalies.
- The larger table is split into smaller tables during normalisation and these tables are connected through relationships.
- Redundancy in the database table is reduced by using the normal form.

Process

Database normalisation is a step-by-step, formal method that enables us to break down database tables such that update anomalies and data reliance are both reduced to a minimum. In order to analyse the tables, it uses the primary key or candidate key and any functional dependencies that may be present in the table. First Normal Form (INF), Second Normal Form (2NF) and Third Normal Form were the initial names given to the suggested normal forms (3NF).

Boyce-Codd Normal Form, which was later introduced by R. Boyce and E. F. Codd, is a more robust definition of 3NF. All of these normal forms, with the exception of 1NF, are based on the functional dependence between table attributes. Later, higher normal forms than BCNF, such as the Fourth Normal Form (4NF) and Fifth Normal Form, were introduced (5NF). These later typical forms, however, deal with extremely uncommon circumstances.



1St Normal Form

- If an atomic value is present in a relation, it will be 1NF.
- It states that a table's attribute cannot have more than one value. It can only contain attributes with a single value.
- The multi-valued attribute, the composite attribute and their combination are not allowed in the first normal form.

Notes

Example: Due to the multi-valued parameter EMP_PHONE, relation EMPLOYEE is not in 1NF.

Employee Table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

The EMPLOYEE table has been broken down into 1NF as shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

2nd Normal Form

- Relational must be in the 1NF in the 2NF.
- In the second normal form, the primary key is necessary for the complete functionality of all non-key attributes.

Example: Let's say that a school has a database where instructors' information and the subjects they teach are stored. A teacher in a school is allowed to teach multiple subjects.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

The non-prime attribute TEACHER_AGE in the provided table is reliant on TEACHER_ID, a suitable subset of a candidate key. Because of this, it breaks the 2NF rule.

We divide the provided table into two tables in order to transform it into 2NF:

TEACHER_DETAIL table:

Notes

TEACHER_ID	TEACHER_AGE
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Math
83	Computer

3rd Normal Formal

- If a relation is in 2NF and does not have any transitive partial dependencies, it will be in 3NF.
- The amount of duplicate data is decreased with 3NF. Moreover, it is employed to ensure data integrity.
- The relation must be in third normal form if non-prime characteristics do not have transitive dependencies.

If at least one of the following conditions is true for any non-trivial function dependency X →Y, a relation is said to be in third normal form.

1. Super key X is used.
2. Y is a prime property, meaning that every element of Y is a potential key.

Example:

EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

{EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so on

Candidate key: {EMP_ID}

Notes

Non-prime attributes: Except for EMP_ID, every attribute in the provided table is a non-prime. Thus, EMP_ZIP is dependent on EMP_ID, while EMP_ZIP is dependant on EMP_STATE & EMP_CITY. The super key (EMP_ID) is transitively reliant on the non-prime characteristics (EMP_STATE, EMP_CITY). It disobeys the third normal form rule.

To relocate EMP_CITY and EMP_STATE to the new EMPLOYEE_ZIP> table, which has EMP_ZIP as a primary key, is necessary for this reason.

employee desk

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

4th Normal Formal

A level of database normalisation known as the fourth normal form (4NF) is one in which a candidate key is the only non-trivial multivalued dependency. It advances the Boyce-Codd Normal Form and the first three normal forms (1NF, 2NF and 3NF) (BCNF). It specifies that a database shall not have more than one multivalued dependency in addition to meeting the BCNF standards.

Example: Student

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

Notes

The STUDENT table that is being provided is in 3NF, however the COURSE and HOBBY are two separate entities. Because of this, there is no connection between COURSE and HOBBY.

A student with the STU_ID of 21 has two courses—computer and math—as well as two extracurricular activities—dancing and singing. As a result, there is a Multi-valued dependency on STU_ID, which causes data to be repeated unnecessarily. In order to convert the aforementioned table into 4NF, we can divide it into two tables:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

5th Normal Formal

- If a relation is in 4NF, lacks join dependencies and requires lossless joining, it is in 5NF.
- In order to prevent redundancy, 5NF is satisfied when all of the tables are divided into as many tables as possible.
- PJ/NF, or Project-join normal form, is another name for 5NF.

Example:

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

Notes

According to the table above, John enrols in both computer and math classes for semester 1, but not for semester 2. To identify a valid data in this situation, a combination of all these fields is needed.

Consider a scenario in which we add a new semester as Semester 3 but are unsure of the subject or the students who will be enrolled in it, so we leave Lecturer and Subject as NULL. We cannot, however, leave the other two columns empty because the three columns together function as a primary key.

In order to convert the aforementioned table into a 5NF, we can divide it into the following three relations:

P1

SEMESTER	SUBJECT
Semester 1	Computer
Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	LECTURER
Computer	Anshika
Computer	John
Math	John
Math	Akash
Chemistry	Praveen

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

Summary

- A functional dependency between two columns, X and Y, means that for any two records R1 and R2 in the table, if field X of record R1 contains value x and field X of record R2 contains the same value x, then if field Y of record R1 contains the value

Notes

y, then field Y of record R2 must contain the value y. We can say that attribute Y is determined by attribute X.

- Functional dependencies may exist between multiple source columns.
- A functional dependency establishes a relationship between two sets of attributes.
- A transitive dependency exists when you have the following functional dependency pattern:
  - ❖ A→B and B→C; therefore A→C
- A transitive dependency therefore exists only when the determinant that is not the primary key is not a candidate key for the relation.
- A second normal form relation that has no transitive dependencies is, of course, automatically in third normal form.
- Database normalisation is a manner of organising the facts inside the database. Normalisation is a scientific approach of decomposing tables to cast off records redundancy(repetition) and unwanted characteristics like insertion, update and deletion anomalies'.
- Normalisation in DBMS is a technique which facilitates produce database systems which can be cost-effective and feature higher protection models.

Glossary

- DBMS: Database Management System
- EOT: End-Of-Transaction
- ACID: Atomicity, Consistency, Isolation, Durability
- UDF: User-Defined Functions
- BLOB: Binary Large Objects
- IoT: Internet of Things
- SQL: Structured Query Language
- The attribute set on the left side of the arrow, X is called Determinant
- The attribute set on the right side, Y is called the Dependent.
- In Trivial Functional Dependency, a dependent is always a subset of the determinant.
- In Non-trivial functional dependency, the dependent is strictly not a subset of the determinant.
- In Multivalued functional dependency, entities of the dependent set are not dependent on each other.
- In transitive functional dependency, dependent is indirectly dependent on determinant.

Check Your Understanding

1. What is the term used for the relation between two qualities that has a set of restrictions?
   a) Join b) Database design
   c) Functional dependencies d) None of the above
2. Which symbol is used to represent functional dependency?
   a) comma b) arrow
   c) colon d) semicolon
3. What is the right side of the production of a functional dependency called?
   a) dependent b) determinant

Notes

c) primary key d) secondary key

4. What are the crucial guidelines for functional dependencies in databases?
   a) reflexive rule b) augmentation rule
   c) transitivity rule d) all of the above
5. Which type of functional dependency occurs when there are numerous independent multivalued characteristics in a single table?
   a) Trivial functional b) Non-trivial functional
   c) Multivalued d) Transitive

Exercise

1. What do you mean by query processing?
2. How to convert sql queries into relational algebra?
3. Define various algorithms for executing query operations.
4. Define:
   a. Parsing and translation
   b. Optimisation
   c. Evaluation
   d. Query tree
   e. Query graph
5. What do you mean by heuristic optimisation of query tree?

Learning Activities

1. Consider a database system designed to store information about a university's course offerings, students and instructors. The initial database design has been provided to you and your task is to normalise the database to a suitable normal form. Justify each step of normalisation, including the identification of functional dependencies, elimination of anomalies and the resulting normalised tables.
2. Consider a scenario where a programming language is designed with its corresponding compiler. The compiler needs to go through the processes of parsing and translation to convert the source code into machine code. Explain the concepts of parsing and translation in the context of compiler design, addressing their significance, challenges and the overall impact on the compilation process.

Check Your Understanding- Answers

1. c 2. b 3. a 4. d
5. c

Module - II: Object Oriented and Extended Relational Database Technologies

Learning Objectives

At the end of this module, you will be able to:

- Understand object-oriented database
- Analyse the architecture of object-relational database management system
- Analyse architecture of object-oriented database management system
- Define object-oriented and relational design modelling
- Understand aggregation and associations
- Define the object query language and object-relational concepts

Introduction

It appears that relational and object-oriented databases are engaged in a never-ending struggle for greater storage power in the field of database administration. To determine which database type has the most potential for your data management requirements, let's explore the worlds of relational and object-oriented databases.

Databases are used to efficiently and methodically store and manipulate data. Relational and object-oriented databases are two of the various types of databases. Relational databases and object-oriented databases are two very potent methods for handling data, but they handle information in different ways.

A relational database employs structured query language (SQL) to retrieve information and arranges data into tables. A relational database has different tables, each with fields holding distinct data. These fields, or columns, are all connected to one another in some manner and make up a single table. This makes it simple to retrieve data from several sources with only one query.

Because of the thorough organisation that makes it possible to efficiently organise every piece of information and build relationships between table parts, relational databases are able to store data more accurately. Compared to traditional techniques like file-based systems, which become inefficient when the quantity of records increases dramatically, relational databases are extremely scalable, ensuring improved performance.

Using techniques from object-oriented programming, object-oriented databases—also referred to as object databases—are a subset of database management systems (DBMS) that store data as objects. An object in an object-oriented or entity-oriented database is a group of methods and properties that describe the object. While the methods specify behaviours or actions, the attributes define states or properties. With this kind of database, you can write and run queries with more flexibility since you may address other characteristics of the objects in the database, such as the States and Actions, instead of just the data values.

An object-oriented database, as opposed to a relational database, combines the ideas of behaviour and data structure into individual objects, giving users more direct access to their data because they are interacting with objects rather than tables and columns. Because objects have pertinent methods to interact with on each platform, they also

Notes

enable apps to incorporate data fast. Large enterprises and private clouds both use object-oriented databases, primarily because of their capacity to handle massive volumes of both structured and unstructured data.

2.1 Object-Oriented Databases and Their Architecture

Object Manager, Query Processor and Storage Manager are some of the components that make up the architecture of object-oriented databases (OODBs), which are database management systems that follow the principles of object-oriented programming. These components allow for the direct representation of complex data structures in the database and facilitate seamless integration with object-oriented programming languages.

Concepts from object-oriented programming are closely related to those in object-oriented databases. The following are object-oriented programming's four key concepts:

Polymorphism: An object that is polymorphic can assume different forms. This feature makes it possible for diverse data kinds to be used with the same programme code. Bicycles and cars can break, but they do so through distinct mechanisms. The action break in this instance is a polymorphism. The defined action is polymorphic, meaning that the outcome varies based on the vehicle that executes.

Inheritance: While making some code reusable, inheritance establishes a hierarchical link between related classes. When new types are defined, all of the current class fields and methods are inherited and further extended. The parent class is the one that currently exists and the child class expands upon it.

Encapsulation: The capacity to combine mechanisms and data into a single object for access control is known as encapsulation. This procedure results in data and function security by hiding specifics and bits of information about how an item functions. Through methods, classes communicate with one another without requiring knowledge of how specific methods operate.

Abstraction: Representing only the necessary data aspects for the required functionality is known as abstraction. Important information is chosen by the process and irrelevant data is kept hidden. Reusability and a reduction in complexity are two benefits of abstraction for modelled data.

2.1.1 Object-Oriented Database

It was first suggested in the late 1960s to design software using an object-oriented methodology. But it took over 20 years for object technologies to catch on. Popularity with object-oriented methods increased in the 1980s and 1990s. A growing number of information systems and engineering professionals, as well as many software product manufacturers, adopted the object-oriented approach as their paradigm of choice during the 1990s. These days, traditional methods for database architecture and software development have gradually been supplanted by object technologies.

Typically, applications that leverage strong modelling techniques, intuitive graphical user interfaces (GUIs) and sophisticated data management capabilities are linked to object-oriented database (OODB) systems. We will go over the main ideas of object-oriented databases (OODBs) in this unit. We will also talk about object-oriented languages utilised in object-oriented database management systems (OODBMSs).

Dr. E.F. Codd initially developed the relational data model in his groundbreaking study, which addressed the drawbacks of traditional database techniques like hierarchical and network (CODASYL) databases. Since then, over a hundred commercial relational

Notes

database management systems have been created and implemented for usage in PC and mainframe settings. RDBMSs do, however, have many drawbacks, chief among them being their restricted modelling capabilities. In order to create database designs that more closely resemble the "realworld," several data models were created and put into use. Figure below illustrates the development of data models.

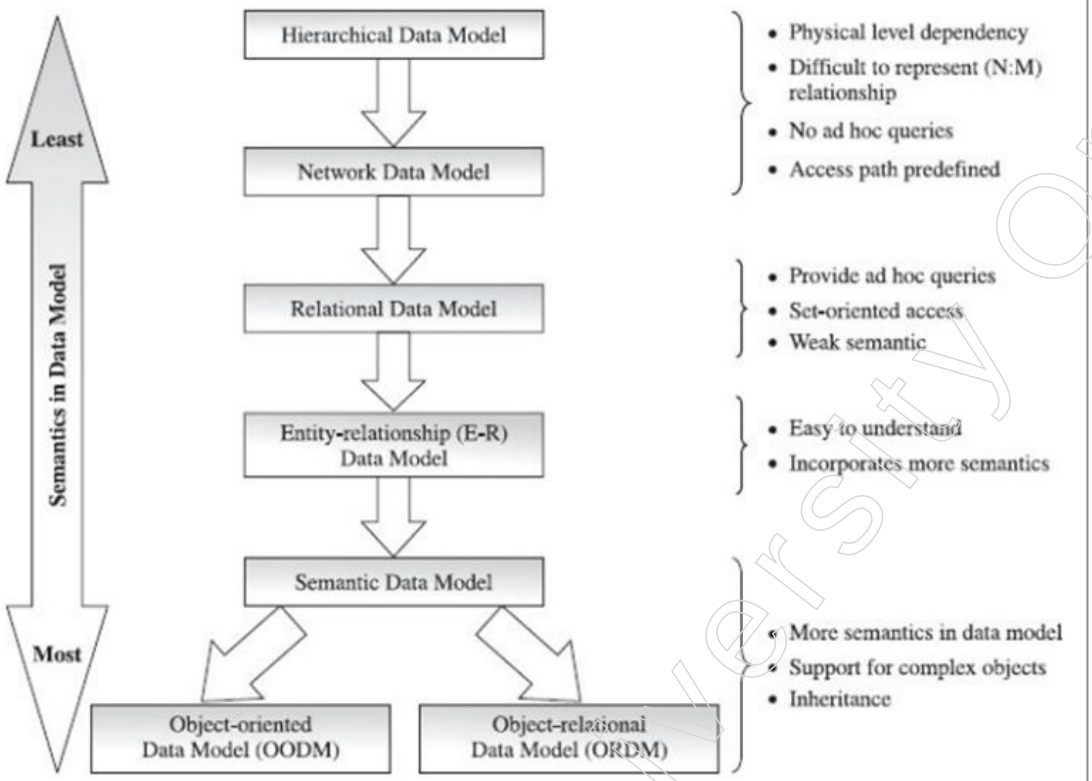


Figure: History of evolution of data model

Image Source: Image Source: Database Management System, S K Sinha, Second Edition

Every data model made use of the flaws in earlier models. A network model took the role of the hierarchical model since it made it much simpler to depict complicated (many-to-many) relationships. Conversely, the relational model's improved data independence, easier-to-use query language and simpler data representation provided a number of advantages over the hierarchical and network models. Chen then presented the entity-relationship (E-R) model, which provides an intuitive graphical data representation. The database design standard is now the ER model. A distinct data model was required to accurately reflect the real world when more complex real-world issues were simulated. In an effort to extract additional meaning from real-world items, M. Hammer and D. McLeod developed the Semantic Data Model (SDM). More semantics were added to the data model by SDM, along with new ideas like class, inheritance and other ideas. This made the models of the actual objects more realistic. Two new data models have arisen in response to the growing complexity of database applications:

- Object-oriented data model (OODM).
- Object-relational data model (ORDM), also called extended-relational data model (ERDM).

Third-generation DBMSs are represented by object-oriented data models (OODMs) and object-relational data models (ORDMs). Logical data models that represent the semantics of objects supported by object-oriented programming are called object-oriented data models, or OODMs. OODMs can reflect complexities that relational systems are

Notes

unable to handle and directly implement conceptual models. Many of the ideas created initially for object-oriented programming languages (OOPLs) have been adopted by OODBs. Transient objects are those that exist just while a programme is running in an OOPL. The OODM origins, drawn from many regions, are depicted in the figure below.

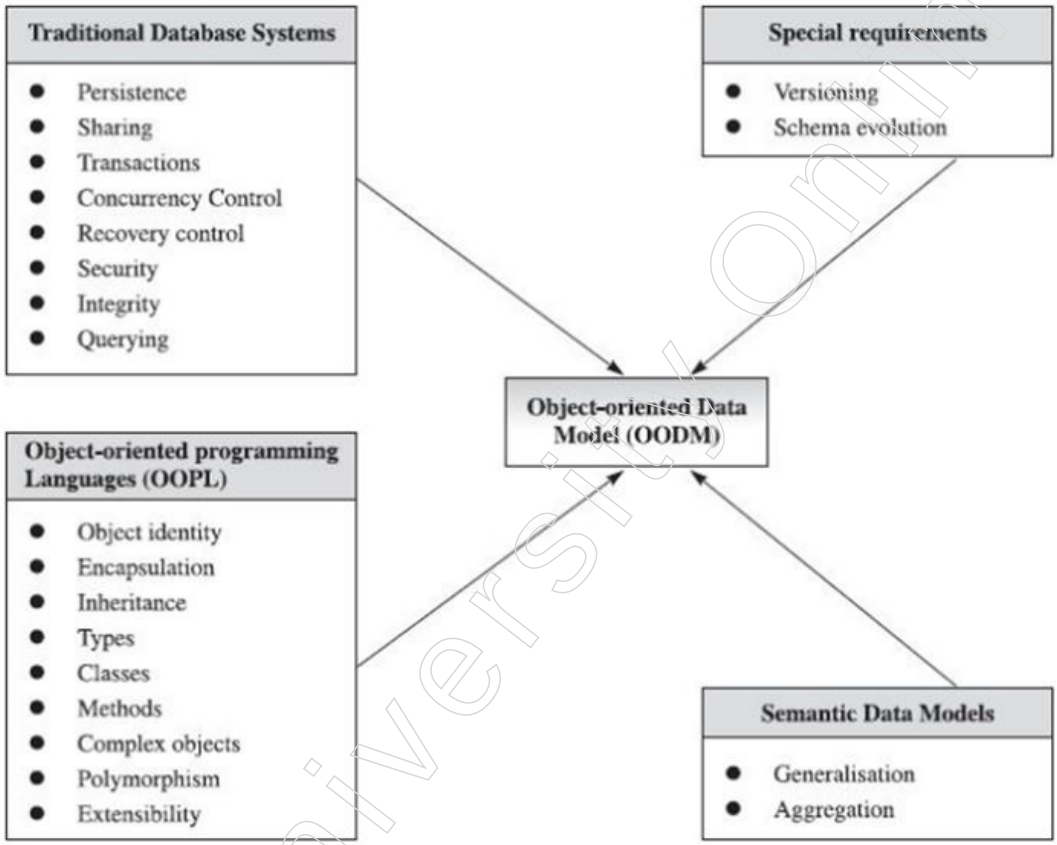


Figure: Origins of OODM

Image Source: Database Management System, S K Sinha, Second Edition

A durable and shareable collection of objects specified by an OODM is called an object-oriented database (OODB). Objects can have an extended life span in an OODB, allowing for persistent storage. Because of this, the objects survive programme termination and can be shared and retrieved at a later time. Or, to put it another way, OODBs enable the sharing of persistent objects by several programmes and applications while permanently storing them on secondary storage (discs). One or more OOPLs are interfaced with by an OODB system to offer shared and permanent object functionality.

Characteristics of Object-oriented Databases (OODBs)

- Ensure that items in databases and the actual world have a clear correlation in order to prevent objects from losing their identity or integrity.
- To make it simple to identify and interact with an object, OODBs assign each object a distinct system-generated object identifier (OID). In contrast, every relation in the relational model needs to have a primary key property, the value of which uniquely identifies every tuple.
- Because OODBs can define new data types and the operations to be done on them, they are expandable databases.
- Support encapsulation, in which the methods' implementation and data representation are concealed from outside parties.

- An object that exhibits heredity is one that takes on the characteristics of other objects.

Comparison of an OODM and E-R Model

Notes

The table below compares an entity-relationship (E-R) model with an object-oriented data model (OODM).

OODM and conceptual data modelling (CDM), which are founded on entity-relationship (E-R) modelling, differ primarily in that OODM objects encompass both state and behaviour. In contrast, CDM is simply able to record states and is unaware of behaviour. Since CDM lacks a notion of communications, it is unable to implement encapsulation.

Table: Comparison between OODM and ERDM

Image Source: Database Management System, S K Sinha, Second Edition

SN	*OO Data Model*	*E-R Data Model*
1.	Type	Entity definition
2.	Object	Entity
3.	Class	Entity set/super type
4.	Instance Variable	Attribute
5.	Object identifier (OID)	No corresponding concept
6.	Method (message or operations)	No corresponding concept
7.	Class structure (or hierarchy)	E-R diagram
8.	Inheritance	Entity
9.	Encapsulation	No corresponding concept
10.	Association	Relationship

An OODB is managed by an object-oriented database management system (OODBMS). Subsets of the OODM functionalities are used by numerous OODBMSs. As a result, individuals who design OODBMSs typically choose OO features—like support for single or multiple inheritances and early or late binding of data types and methods—that best fulfil the goals of the OODBMS. Numerous OODBMSs have been used in both commercial and research purposes.

Features of OODBMSs

Object-oriented (OO) Features:

- Needs to accommodate complicated things.
- Object identification must be supported.
- Has to facilitate encapsulation.
- Needs to accommodate classes or types.
- Classes and types need to be able to inherit from their forebears.
- Dynamic binding must be supported.
- There must be computational completion in the data manipulation language (DML).
- An expandable set of data types is required.

General DBMS Features:

- It is necessary to provide data permanence, which entails having the ability to recall data locations.

Notes

- Needs to be able to handle very big databases.
- Needs to accommodate multiple users at once.
- Must be able to recover from malfunctions in both software and hardware.
- A straightforward data query is required.

Advantages of OODBMSs

- Improved modelling capabilities: It makes it possible to model the real environment more precisely.
- More extensibility and less redundancy are achieved by enabling the construction of new data types from preexisting ones. It can factor out shared characteristics between several classes and create superclasses from them that subclasses can share.
- Impedance mismatch elimination: It offers a single language interface for programming languages and data manipulation languages (DMLs).
- More eloquent wording for queries: As opposed to SQL's associative access, it offers navigational access for data access between objects.
- Support for schema evolution: Schema evolution is more practical in OODBMSs. The schema can be made more understandable, more structured and able to encapsulate more application semantics thanks to generalisation and inheritance.
- Support for long-duration transactions: OODBMSs handle this type of transaction using a different protocol than RDBMSs, which enforce serialisability on concurrent transactions in order to maintain database consistency.
- Applications for advanced database systems: Because of the OODBMSs' enhanced modelling features, advanced database systems, including computer-aided design (CAD), office information systems (OIS), computer-aided software engineering (CASE), multimedia systems and so on, can benefit from using them.
- Enhanced output. It enhances the DBMS's overall performance.

Disadvantages of OODBMSs

- Absence of a common data model.
- insufficient experience.
- Absence of guidelines.
- RDBMS and the newly-emerging ORDBMS systems are competitors.
- Encapsulation is compromised by query optimisation.
- Performance may be affected by object-level locking.
- complexity brought forth by an OODBMS's enhanced capabilities.
- Absence of backing for opinions.
- Absence of backing for safety.

2.1.2 OO Concepts

A set of design and development guidelines known as object-orientation is predicated on the idea of conceptually independent computer structures, or objects. Every object is a representation of a real-world thing that may communicate with other objects as well as with itself. Our world is made up of things. These things can be found in nature, in things created by humans, in commerce and in the goods we use on a daily basis. They are able to be made, merged, classified, explained and arranged. Consequently, we can model the world using an object-oriented perspective to make it easier for humans to comprehend and traverse.

Notes

The issue domain of an object-oriented (OO) system is defined as a collection of objects with particular characteristics and behaviours. A set of functions, referred to as operations, services, or methods, are used to manipulate the objects and a messaging protocol is used to facilitate communication between them. Classes and subclasses are used to categorise objects. Reuse is facilitated by OO technologies and reuse speeds up software design and development.

Object-oriented programming languages (OOPLs) were created as an alternative to conventional programming techniques and this is where object-oriented principles originate. The first programming languages to use OO notions were Ada, Algol, LISP and SIMULA. Eventually, C++ and Smalltalk emerged as the two most popular object-oriented programming languages (OOPLs). These days, databases, software engineering, knowledge bases, artificial intelligence and computer systems in general all use OO ideas.

Objects

An object is an abstract representation of a real-world thing with embedded properties, a distinct identity and the capacity to interact both with other objects and with the real world. It is a singularly identifiable entity that possesses both the characteristics that characterise an actual object's state and the actions connected to it. A name, a collection of characteristics and a set of functions are all possible for an object. An object might be an individual or it can be a member of a class of related objects. Consequently, a description of an object's characteristics, behaviours, identification, operations and messages are all included in its definition. Data and the processing done on it are both contained in an object.

Two components make up a typical object: behaviour (operations) and state (value). As a result, it resembles a programme variable in a programming language to some extent, but it usually has a sophisticated data structure and particular operations that the programmer has defined. Figure below provides examples of the objects. Rectangles are used to symbolise each object. The object's name appears as the first item in the rectangle. A straight line divides the object name from the object properties. There can be zero or more characteristics on an object. Every characteristic has a unique name, value and set of requirements. A list of services or actions comes after the attributes. Every service has a name, which will eventually be converted into machine code that may be executed. A horizontal line divides the list of properties from the services or activities.

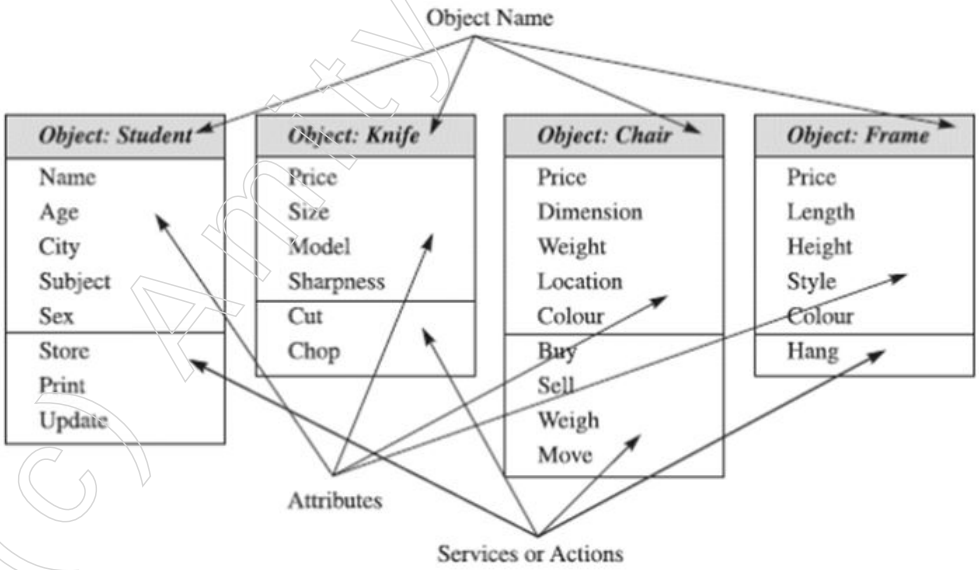


Figure: Examples of objects

Image Source: Database Management System, S K Sinha, Second Edition

Notes

Object Identity

An Object Identifier (OID) is a unique representation of an object's identification. With the OODB technology, every independent object kept in the database has a distinct ODI. An object cannot have more than one OID. The system assigns the OID; it is independent of the values of the object's attributes. The system uses an ODI internally to uniquely identify each object and to construct and maintain inter-object interfaces; the value of an ODI is not accessible to external users. The following traits apply to OID:

- It is produced by the system.
- It is specific to the thing.
- It is unchangeable.
- It cannot be changed while it is alive.
- It cannot be removed. Only when the object is erased can it be removed.
- You can never use it again.
- It is not affected by the values of its characteristics.
- The user is unable to see it.

It is important to distinguish between the OID and the relational database's main key. A primary key in a relational database, as opposed to an OID, is made up of user-defined values for particular attributes that are changeable at any moment.

Object Attributes

Objects in an object-oriented context are defined by their characteristics, or instance variables. Every attribute has a distinct name and corresponding data type. As seen in the above Figure, the object "Student" has qualities like name, age, city, subject and sex. Comparably, the "Knife" object has characteristics such as cost, dimensions, make and model, edge sharpness and so on.

Figure below displays the properties of the objects "Chair" and "Student." It is also possible to employ conventional data types, commonly referred to as base types, such real, integer, string and so forth. Moreover, attributes have domain. The collection of all possible values that an attribute can have is logically grouped and described by the domain. For instance, the real number base data type can be used to represent the possible values of the GPA, as seen in the figure below.

Object: Student	*Attributes*
Name	Abhishek
Age	23
GPA	9.5
Subject	Computer Engg
Sex	Male

Object: Chair	*Attributes*
Price	INR 5000
Dimension	3 FT
Weight	5 KG
Location	Office
Colour	Black

Figure: Attributes of objects 'Student' and'Chair'

Image Source: Database Management System, S K Sinha, Second Edition

Examples of Objects

Type constructors can be used to define the data structures for an OO database schema. The objects "Student" and "Chair" of Below Figure can be specified in a way that corresponds to the object instances, as shown in below figure.

Notes

define type Student;		
tupel 9	Name;	string;
	Age:	float;
	GPa:	float;
	Subject:	string:
define type Chair:	Sex:	chair;
tuple(	Price:	float:
	dimension:	string;
	weight:	float;
	Location:	string;
	colour:	chair;);

Figure: Specifying object types 'Student' and 'Chair'

Image Source: Database Management System, S K Sinha, Second Edition

A class is a group of related objects that have the same behaviours (methods) and structure (attributes). It includes information on how the methods for the objects in that class are implemented as well as a description of the data structure. As a result, every object in a class has the same structure and message response mechanisms. A class also serves as a repository for related things. As a result, a class consists of a class name, a collection of properties and a collection of functions. Class instances, often called object instances, refer to each object within a class. Each class defines two implicit service or action functions: PUT <attribute> and GET <attribute>. The PUT method appends the computed value of the attribute to the attribute name, while the GET function ascertains the value of the linked attribute.

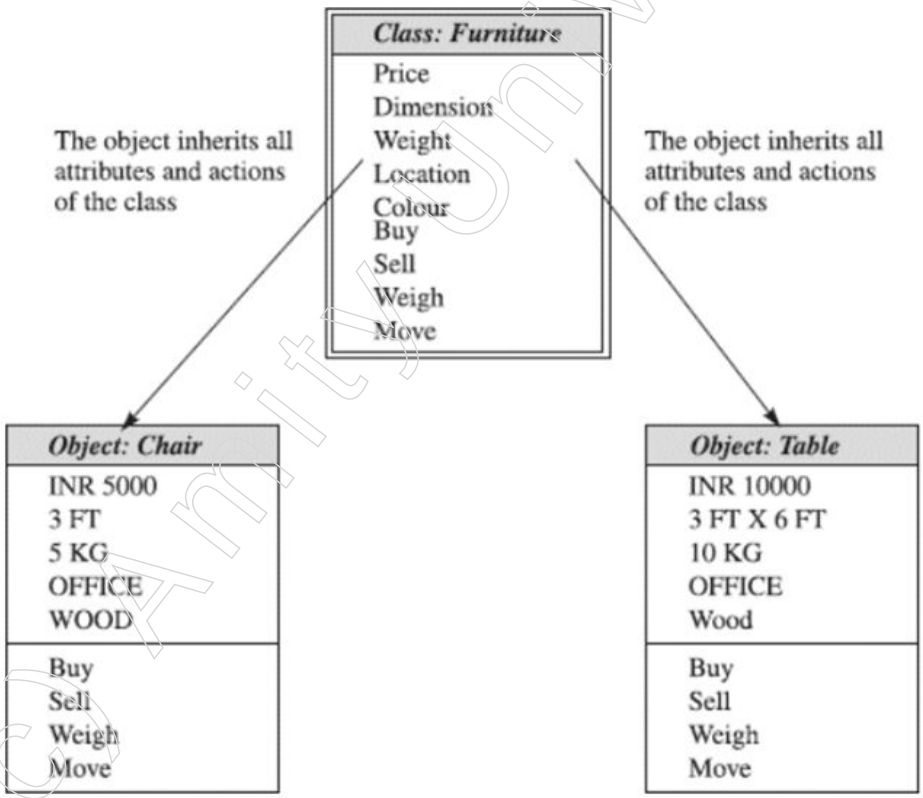


Figure: Example of Class 'Furniture'

Image Source: Database Management System, S K Sinha, Second Edition

Notes

An example of a class called “Furniture” with two instances is shown in the figure below. A member (or instance) of the class “Furniture” is the “Chair.” Every item in the class “Furniture” can have a set of generic properties attached to it, such as cost, size, weight, placement and colour. As “Chair” belongs to the class “Furniture,” all of the class’s declared attributes are passed down to it.

When new instances of the class are generated, the attributes can be reused once the class has been specified. As an illustration, let’s say that a new object named “Table” has been defined and belongs to the class “Furniture,” as seen in the figure below. “Table” acquires all of the characteristics of “Furniture.” The services related to the class “Furniture” are move (moving the furniture object from one location to another), sell (selling the furniture item) and buy (buying the furniture object).

Another example of a class called “Student” with three instances—”Abhishek,” “Avinash,” and “Alka”—is shown in the figure below. Every instance is an object that is a part of the “Student” class. All of the characteristics and functions of the “Student” class are present in every instance of the “Student” object. This class’s associated services are Store, Print and Update. Store writes the student object to a file. Prints the object attributes. Updates one or more attributes by changing their value.

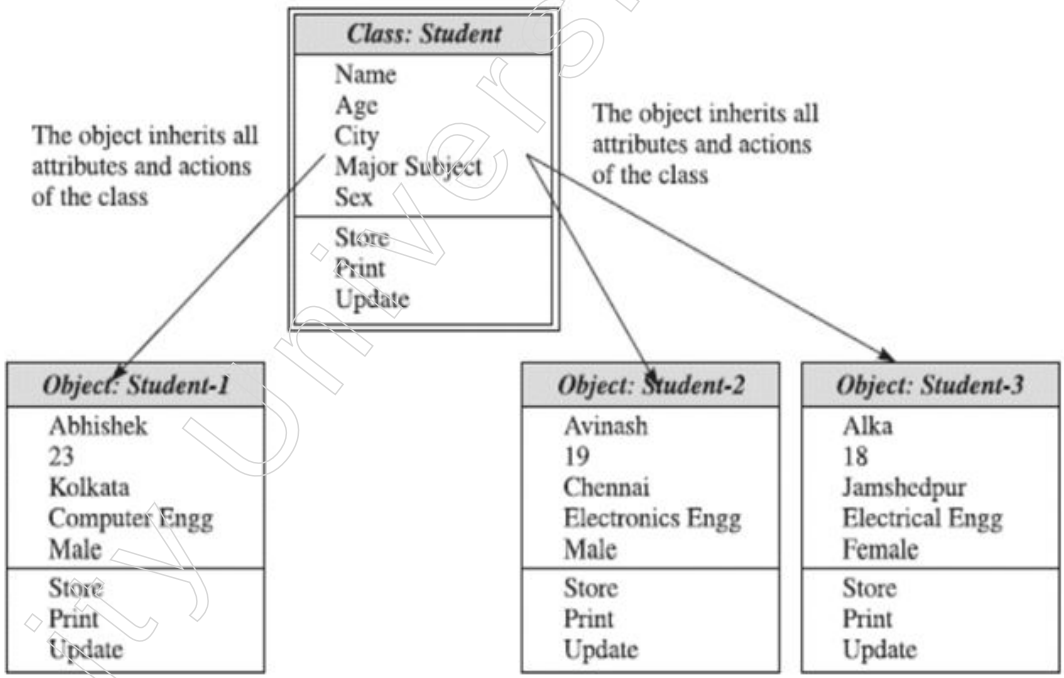


Figure: Example of Class ‘Furniture’

Image Source: Database Management System, S K Sinha, Second Edition

Another example of a class “Employee” with two instances, “Jon” and “Singh,” is shown in the figure below. Every instance is an object from the “Employee” class. Every “Employee” instance is an object with all of the characteristics and functions of the “Employee” class. This class’s associated services are Print, which outputs the attributes of the object and Update, which updates the values of one or more attributes. A single-lined rectangle is used to represent an object and a double-lined rectangle is used to represent the class. A double-line could indicate that there are several objects in the class.

Notes

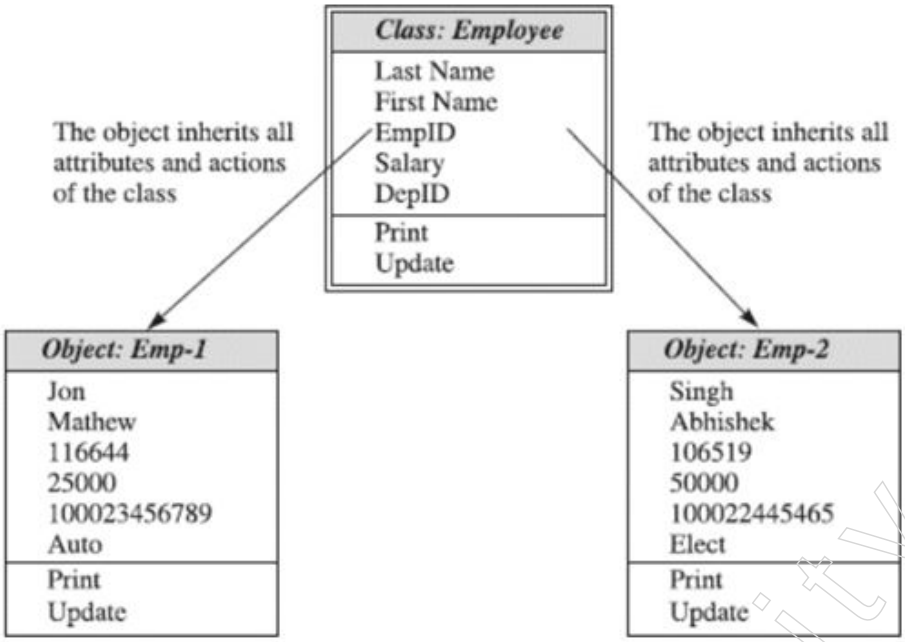


Figure: Example of classes 'Employee'

Image Source: Database Management System, S K Sinha, Second Edition

Relationship or Association among Objects

A given connection can relate two objects, either of the same class or of different classes. To correlate specific class objects with one another, we may define the association "Same-Major Subject" in the "Student" class, for instance. As a result, two objects are connected if and only if the values of their "Major Subject" attributes match. Stated differently, two students who are enrolled in the same "Major Subject" have a relationship or connection to one another.

One-to-one (1:1), one-to-many (1:N), many-to-one (N;1), or many-to-many (N:M) associations can exist between classes or objects. A line joining the related classes or objects together illustrates the link between them graphically.

A multiplicity number (or range of numbers) may be introduced and printed below the association line to represent the association multiplicity number for associations that are not one-to-one (1:1). The name of the association may be written on top of the relation line if it is not evident from the context or if there are several associations between the items. The two names of a relation whose inverse is given a separate name can appear on either side of the association line.

The linkages or relationships between classes are shown in the figure below. The link between the classes "Student" and "Course" is depicted below in Figure (a). In this instance, the relationship—which is one-to-many (1:N)—between two classes is referred to as "enrolled." This implies that a student might have zero or more courses registered for them.

The relationship between the classes "Student" and "Advisor" in Figure (b) below is known as "Advisee-of" and is many-to-one (N:1) in nature. The one-to-many (1:N) link known as "Advisor-of" exists between the classes "Advisor" and "Student." The relationship between the classes "Student" and "Sport_Team" is referred to as "Member-of" in Figure (c) below, while the relationship between the classes "Sport_team" and "Student" is referred to as "Team." These two relationships are many-to-many (N:M).

Notes

```
class Employee {
        Private
                char    LastName [20+1];    //    employee last name
                char    FirstName [20+1];   //    employee first name
                char    EmpID [5+1];        //    employee Identifier
                float   Salary;             //    employee salary
                char    SSN [11+1]          //    social security number
                char    DepID [5+1]         //    department ID
                void set_EmpID (char *Emp_ID) {strcopy (EmpID, Emp_ID);}
                void set_DepID (char *Dep_ID) {strcopy (DepID, Dep_ID);}
        Public
                Employee (char *Dep_ID, char *Emp_ID);
                    {
                    set_EmpID (Emp_ID);
                    set_DepID (Dep_ID);
                    }                       //    in-line function
                Employee (char *Dep_ID, float empSalary, char  *Emp_ID);
                    {
                    set_EmpID (Emp_ID);
                    set_DepID (Dep_ID);
                    set_Salary (sal);
                    }
                ~ Employee ();              //    in-line function
                void set_LastName (char *LName) {strcopy (LastName, LName);}
                void set_FirstName (char *FName) {strcopy {FirstName, FName);}
                void set_EmpID (char *Emp_ID) {strcopy (EmpID, Emp_ID);}
                void set_Salary (float sal) {Salary = sal;}
                void set_SSN (char *soc_sec) {strcopy (SSN, soc_sec);}
                char *get_LastName (void) return & LastName [0];}
                char *get_FirstName (void) return & FirstName [0];}
                char *get_EmpID (void) return & Emp ID [0];}
                char *get_Salary (void) return Salary;}
                char *get_SSN (void) return & SSN [0];}
                char *get_Dep_ID) (void) return & DepID [0];}
};
```

Figure: Implementation of class 'Employee' using C++

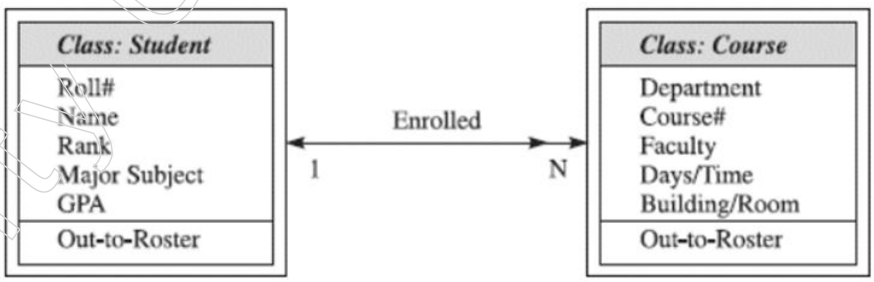


Figure-(a) : Association among classes

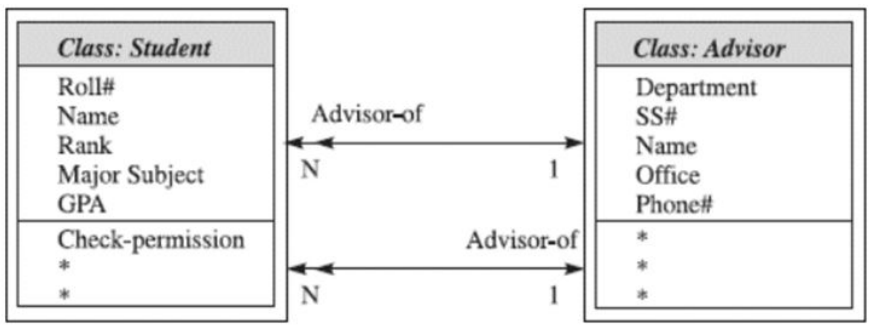


Figure-(b) : Association among classes

Notes

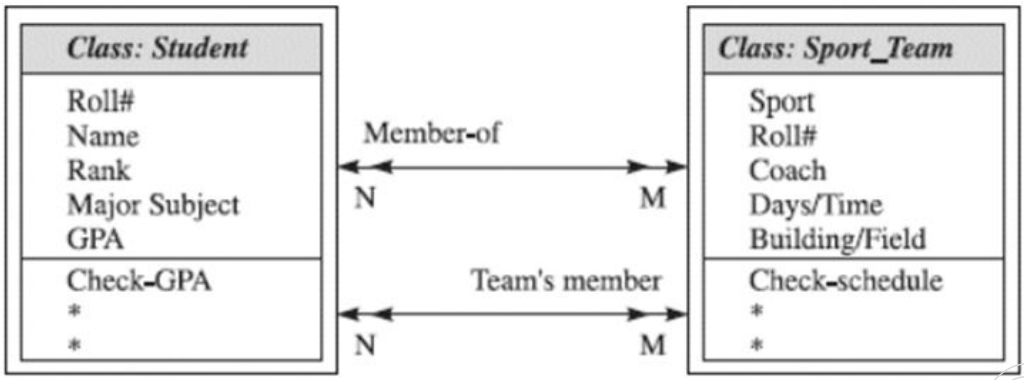


Figure-(c) : Association among classes

Image Source: Database Management System, S K Sinha, Second Edition

Structure

In essence, structure is the relationship between a class and its objects. Let's think about the subsequent classes:

a) Person

b) Student

c) Employe

d) Graduate

e) Undergraduate

f) Administration

g) Staff

h) Faculty

The following observations can be drawn from the aforementioned classes:

- Graduate and undergraduate are each a subclass of 'Student' class.
- Administrator, staff and faculty are each a subclass of 'Employee' class.
- Student and employee are each a subclass of 'Person' class.

Inheritanceis transferring the superclass's characteristics to each of its subclasses. It is the capacity of an object to inherit the behaviours (methods) and data structure of the classes above it from other objects within the hierarchy or structure.

The modelling technique known as inheritance in databases allows a new table (subclass) to inherit characteristics, relationships and features from an existing table (superclass). This allows for the reuse of structure and attributes by mirroring the 'is-a' relationship between entities. By inheriting traits from the superclass, the subclass builds a hierarchy that improves abstraction and representation of real-world relationships and makes data organisation easier.

Two categories of inheritance exist:

a. Single inheritance: When a class has just one immediate (parent) superclass above it, single inheritance is present. Classes "Student" and "Employee" inheriting the immediate superclass "Person" is an example of a single inheritance.

b. Multiple inheritances: When a class derives from multiple parent superclasses that are directly above it, multiple inheritances are present.

Notes

Operation

An operation is a service or function that every instance of a class provides. Only via these kinds of actions can information contained within an object be accessed or changed by other objects. As a result, the action offers a class an external interface. Without revealing the core workings of the class or how its functions are carried out, the interface only displays the outside of it. The following four categories apply to the operations:

a. Constructor operation: The constructor operation generates a new class instance.

b. Query operation: It retrieves an object's state without changing it. There are no negative effects.

c. Update operation: This operation modifies an object's state. There are adverse effects.

d. Scope operation: This action pertains to classes as opposed to individual object instances.

Polymorphism

Systems that are object-oriented allow for operation polymorphism. Operator overloading is another term occasionally used to describe the polymorphism. Depending on the kind of objects the operator is applied on, the concept of polymorphism permits the same operator name or symbol to bind to two or more distinct operator implementations.

Advantages of OO Concept

Many computer-based professions have made extensive use of OO concepts, particularly those that include intricate programming and design issues. The benefits of OO concepts to numerous computer-based disciplines are presented in the table below.

SN	Computer-based Discipline	OO advantages
1	Programming languages	Easier to maintain. Reduces development time. Enhances code reusability. Reduces the number of lines of code. Enhances programming productivity
2	Graphical User Interface (GUI)	Improves system user-friendliness. Enhances ability to create easy-touse interface. Makes it easier to define standards.
3	Design	Better representation of the realworld situation. Captures more of the data model's semantics.
4	Operating System	Enhances system probability. Improves systems interoperability
5	Databases	Supports complex objects. Supports abstract data types. Supports multimedia databases.

2.1.3 Architecture of Object-Relational Database Management System

A sort of database management system that combines the characteristics of relational and object-oriented databases is called an object-relational database management system, or ORDBMS. It combines the relational database model with ideas from object-oriented

programming, like polymorphism, inheritance and encapsulation. The architecture of an object-relational database management system is summarised as follows:

Notes

Relational Database Foundation: The relational model serves as the basis for designing tables, relationships and constraints. An ORDBMS begins with the construction of a conventional relational database system, which consists of tables with rows and columns to store data.

Object-Oriented Extensions: Object-oriented features are added to the relational model by ORDBMS. It presents new data kinds, including as structures, arrays and other non-scalar types; they are frequently referred to as complex types or user-defined types. Encapsulation, inheritance and polymorphism are examples of objects that are incorporated into the database model.

User-Defined Types (UDTs): With the help of User-Defined Types (UDTs), which are defined by the user and can encapsulate both data and operations, ORDBMSs enable more accurate modelling of real-world entities.

Inheritance: Tables can inherit attributes and functions from other tables thanks to ORDBMS support for inheritance. This makes it possible to create a hierarchy of tables that more accurately depicts the relationships between various elements.

Methods and Functions: Attaching functions or methods to user-defined types is permitted by ORDBMSs. These techniques work with the data contained in the UDT and offer a means to include behaviour in addition to data.

Query Language Enhancements: The object-relational model adds new functionalities, which are supported via extensions to SQL (Structured Query Language). New syntax for querying and modifying intricate data kinds and relationships is frequently a part of this.

Indexing and Performance: Through the implementation of indexing techniques appropriate for complicated data types, ORDBMSs maximise performance. Indexes aid in accelerating data retrieval, particularly when working with features that are object-oriented.

Integration with Programming Languages: Programming languages that enable object-oriented programming, like Java or C++, can be seamlessly integrated with ORDBMS using interfaces, or APIs (Application Programming Interfaces).

Transaction Management and Security: Transaction management and security capabilities are included in ORDBMSs, just like in conventional relational databases, to guarantee data integrity and safeguard critical information.

The ORDBMS's last responsibility after selecting a query plan is to carry out each of the plan's steps in the right order and provide the results back to the user who submitted the query. Occasionally, there may be a significant delay between the submission of a query to the ORDBMS and its execution. Long before the results of the query are needed, queries can be prepared. We should split the ORDBMS into two functioning parts because of this circumstance. The query-processing duties covered in the preceding sections are carried out by the first, top half of the ORDBMS engine. These jobs include parsing the query string into an internal form, optimising this internal form into a plan and scheduling the execution of the optimised plan. All of the low-level data management functions, including JOIN, SORT and RESTRICT, are handled by the second, bottom half of the ORDBMS. The top half of the ORDBMS uses features built in the bottom half to carry out each operation in a query plan, including memory management, scheduling, low-level operations, buffer management (I/O and caching) and transaction support (locking, logging, etc.).

Notes

This perspective on the ORDBMS emphasises the magnitude of the modifications required to convert an RDBMS into an ORDBMS. Similar to how the ORDBMS's sorting algorithms are generalised, all of the functional components labelled in the figure below must also be generalised. In addition to these changes, the engine's list of data management algorithms and techniques has to be expanded in order to effectively manage specific types of components. For instance, an R-tree is needed for indexing spatial data and more recently, a number of academics have created join computation algorithms for spatial data that are more effective than the straightforward nest loop. Furthermore, the justification for join indices and other approaches is stronger than it was for SQL-92 DBMSs due to joins using costly predicates.

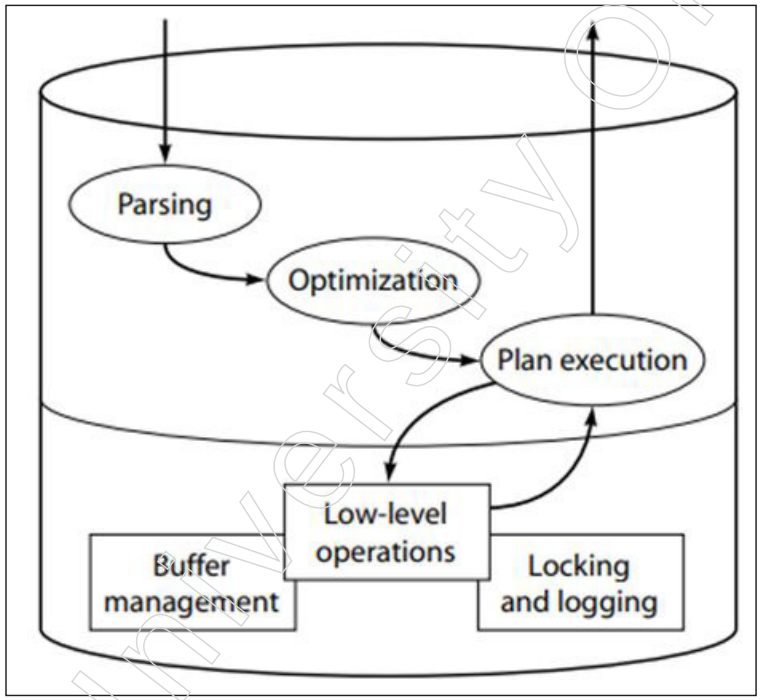


Figure: Internal architecture of ORDBMS

Image Source: Component Database Systems, Klaus R. Dittrich andreas Geppert

2.1.4 Architecture of Object-Oriented Database Management System

Using the concepts of object-oriented programming, an object-oriented database management system (OODBMS) is made to store, retrieve and manage data. The architecture of an object-oriented database management system is summarised as follows:

Object Data Model: The object data model, which enables the representation of actual entities as objects, is the foundation of an OODBMS. Data and behaviour are combined into a single unit by objects, which combine methods (procedures or functions) and attributes (data fields).

Object Identity: To differentiate it from other objects, every object in the database has a unique identifier called an object ID, or OID. To preserve relationships between things and guarantee their integrity, object identity is essential.

Classes and Inheritance: Classes are a feature of OODBMSs that act as templates for the creation of objects. Because of inheritance, classes can be arranged in a hierarchy with subclasses gaining methods and characteristics from parent classes.

Complex Data Types:Complex data types like arrays, structures and sets are supported by OODBMS. This makes it possible to model more intricate and realistic data structures.

Notes

Persistence: Because they may be saved in the database and retrieved at a later time without losing their state, objects in an OODBMS are persistent. Object survival is guaranteed for the lifetime of a program's execution by persistence techniques.

Query Language: Object-oriented query languages, such as OQL (Object Query Language) or an object-oriented version of SQL, are commonly used with OODBMS. using the help of these languages, users can retrieve and modify objects using a familiar syntax.

Indexing and Retrieval: Indexing techniques are used by OODBMS to maximise object retrieval. The efficiency of queries can be improved by using indexes that are based on different object properties.

Concurrency Control: OODBMS has features to control multiple users' or applications' simultaneous access to the database. with a multi-user setting, strategies like versioning and locking aid with consistency maintenance.

Transaction Management: Through transaction management, which enables the grouping of several processes into atomic transactions, OODBMS protects data integrity. The maintenance of the ACID (Atomicity, Consistency, Isolation, Durability) attributes ensures the dependability of transactions.

Security: Security features are built into OODBMSs to manage object access and guarantee the integrity and confidentiality of stored data. Mechanisms for user authentication and authorisation are put in place to limit access according to user roles and permissions.

Integration with Programming Languages: Object-oriented programming languages can be seamlessly integrated with OODBMS interfaces or APIs, making it easier to create applications that communicate with the database.

In order to handle data in a database system, architecture is crucial. The rules governing data movement from database server to client and from client to database server and vice versa, are the subject of the architecture design considerations. An additional crucial factor in the data transfer process between the user and the computer is error handling. The machine is made up of tangible components. When the data is being stored in physical memory, power fluctuations may be the cause of the inaccuracy. Users can request data in an object-oriented format since different types of users are not dependent on databases; yet, in an object-oriented database system, data can be stored in multiple formats (Yanchao Wang1 et al.).

They employed object-oriented databases as their middleware, which decreased the amount of effort and time needed for interpretation across several language translations. The data must be loaded and saved by the object-oriented database system (OODBMS) in its internal data formats, such as persistent C++ structures, if that is the option selected for data management. Arie Shoshani, these data formats are incompatible with legacy software. Similarly, C++ programmes cannot easily access data that is contained in conventional table formats. According to Rick Cattell's article, object/relational mapping solutions can be used to integrate object-oriented programming languages with object-oriented database management systems (DBMSs), however there are certain performance and convenience downsides.

There are various varieties of OODBMS architecture. In this discussion, we focus on the six-layer architecture.

Notes

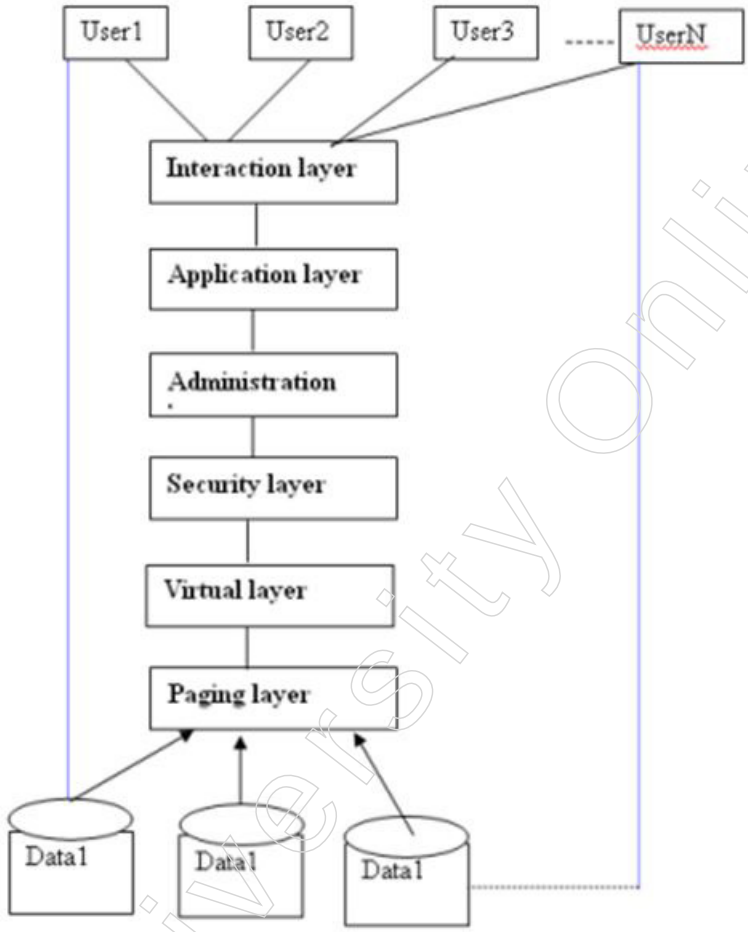


Figure: Six layers Architecture Model for Object oriented database

Image Source : https://www.researchgate.net/publication/266283640_Six_layers_Architecture_Model_for_Object_Oriented_Database/fulltext/54ae4bce0cf24aca1c6f8756/Six-layers-Architecture-Model-for-Object-Oriented-Database.pdf

Interaction Layer

The Six Layers Architecture Model for Object-Oriented Databases starts with the interface layer. Users can communicate with the databases in this layer. Both sending and retrieving data between databases and the user are possible.

Application Layer

In this approach, the application is the second layer. According to Robert Greene's assumption, if an early adopter happened to select an ODB whose architecture wasn't appropriate for their application's demands, the logic would quickly conclude that no ODB could meet their needs. This irrational reasoning gave rise to widespread misconceptions about OODBs, including the claims that they are too slow, incapable of handling high concurrency and unscalable when dealing with massive volumes of data.

Administration Layer

Administrative information management falls under the purview of this layer. This layer has the ability to switch responsibilities as needed. Pharmaceutical businesses, being knowledge-intensive organisations, store a vast amount of scientific documents relevant to their research and development operations. Cristina Ribeiro et al. misused the usual administrative information. Strict quality system requirements also impose document

Notes

workflows and complex data. The quantity of information lost during the preservation process is a critical factor in the success of maintaining complex structures like databases and it is inversely correlated with the amount of metadata supplied in the preservation package. The Administration layer in this paradigm grants authorisation to access the data and regulates its flow.

Security Layer

An essential component of this paradigm is the security layer. The security layer is in charge of giving data complete protection as well as the security of the applications that are used to manage the data. Based on vulnerabilities that were discovered by outside security researchers and subsequently addressed by the manufacturer, David Litchfield compared the security postures of Oracle's RDBMS and Microsoft's SQL server. Both authentication for users and authentication for database administrators can be provided by this layer. In this layer, every security concern is taken into account. Which types of data can be used by whom.

Virtual Layer

Their method's primary benefit is that each slice processor has a very low memory demand that is unaffected by the size of the input. The virtual layer in this paradigm manages the data digitally. This time, a substantial amount of data is handled. The idea of virtualisation is to store data outside of memory. The data are converted in real memory according to the requirements. This approach solves the issue of managing vast amounts of data.

Paging Layer

The division of the data into pages is the responsibility of the paging layer. Managing the pages is simple. The page frame, which divides memory into an equal number of divisions, divides the data into pages of the same size. Large volumes of data can be effectively managed in this way. in an object-oriented database management system are effectively managed. This model was created by taking into account every component of the data. This approach allows data to flow across multiple layers, each of which can carry out its function independently.

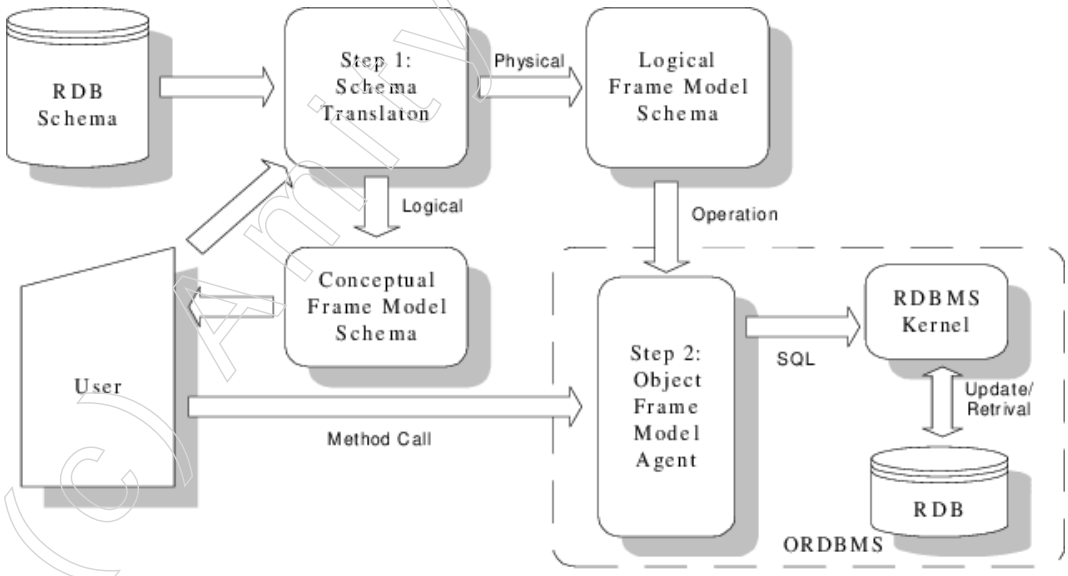


Figure: The ORDBMS Architecture

Notes

2.1.5 Object-Oriented Design Modelling

As the cornerstone of contemporary software engineering, object-oriented design modelling is a thorough and complicated technique that offers an organised method for conceptualising, planning and structuring complex systems using the ideas of object-oriented programming (OOP). This complex method promotes flexibility, extensibility and maintainability in software development by generating abstract models that depict real-world entities and their interactions. The main features of Object-Oriented Design Modelling will be examined in this investigation, including its basic ideas, modelling methods, design patterns and effects on software architecture.

1. Foundational Principles of Object-Oriented Design:

Abstraction:A fundamental idea known as abstraction allows complicated systems to be represented by emphasising key features and disregarding minor elements. Through the construction of abstract classes and interfaces, Object-Oriented Design (OOD) achieves abstraction, enabling developers to more easily model real-world things.

Abstraction can be used, for instance, in a banking system to establish an abstract class called “Account” with shared properties and methods that can then be further subdivided into “SavingsAccount” and “CheckingAccount.”

Encapsulation:Combining data and methods that work with that data into a single unit called a class is called encapsulation. This idea improves information hiding and modularity by blocking direct access to an object’s internal state and permitting regulated interactions via clearly specified interfaces. Encapsulation can be used to encapsulate attributes like name, age and medical history along with methods for updating patient data in a “Patient” class used in a healthcare application.

Inheritance:By taking on traits and behaviours from an already-existing class (superclass), inheritance makes it easier to create a new class (subclass). This encourages the development of a hierarchical structure in OOD, extensibility and code reuse. In a vehicle simulation system, for example, a base class called “Vehicle” can be expanded to provide subclasses like “Car,” “Truck,” and “Motorcycle,” all of which inherit the parent class’s common characteristics.

Polymorphism:It is possible to treat objects of different classes as belonging to the same base class thanks to polymorphism. This idea is made possible in OOD via interfaces and method overrides, which allow for flexibility and adaptability. Polymorphism can be used in a multimedia application to establish a shared interface so that different media kinds (such audio and video) can be played or displayed simultaneously.

2. Modeling Techniques in Object-Oriented Design:

Unified Modeling Language (UML):OOD uses UML, a standardised modelling language, to visually depict system behaviours and architecture. It offers a standardised method of communicating design concepts and contains diagrams like class, sequence and state diagrams. Classes like “Book,” “Author,” and “Library” can be represented in a UML Class Diagram for a library management system, along with their attributes and relationships.

Class Diagrams:Class Diagrams, which show classes, their characteristics and their relationships, are a fundamental modelling approach in OOD. These diagrams act as implementation blueprints for software systems. A class diagram for an e-commerce application might show classes like “Product,” “Order,” and “Customer,” along with the associations that exist between them.

Notes

Sequence Diagrams:Sequence diagrams highlight the way that objects interact with one another and the messages that are transferred during a particular operation or scenario. These are useful diagrams for illustrating a system's dynamic features. A Sequence Diagram can show how a "BankAccount" object and a "Customer" object interact during a fund transfer activity in a banking system.

State Diagrams:State diagrams show the various states in which an object can exist as well as how these states change in response to various occurrences. When modelling systems containing entities that go through different lifespan phases, this modelling technique is quite helpful. A State Diagram might show the states of a "Order" object, such as "Pending," "Shipped," and "Delivered," as well as the transitions between these stages in an online order processing system.

Package Diagrams:Package diagrams group classes and other components into packages, which are modular units. By combining related classes and components, these diagrams aid in the management of system complexity. Package Diagrams can be used in software development environments to group classes pertaining to data access, business logic and user interfaces into distinct packages.

3. Design Patterns in Object-Oriented Design:

Creational Patterns:The process of creating objects is the main emphasis of creational design patterns. Creating objects in a flexible, extendable and consistent manner is made possible by patterns such as Factory Method, Abstract Factory and Singleton. The Singleton pattern can be used in a multimedia application to make sure that all media resources are managed by a single instance of a "MediaLibrary" object.

Structural Patterns:Larger structures are formed by composing classes and objects according to structural design patterns. Class composition flexibility is increased and unambiguous interfaces are defined with the use of patterns like Decorator, Composite and Adapter. The Composite pattern can be applied to a graphical user interface (GUI) framework to treat composite components (complicated dialogues) and individual GUI components (buttons, panels) consistently.

Behavioral Patterns:Object cooperation and communication are the main goals of behavioural design patterns. Observer, Strategy and Command patterns, among others, offer ways to specify how objects interact and carry out certain tasks. The Observer pattern can be used in a messaging system to alert subscribers to new messages, enabling publishers and subscribers to communicate separately.

4. Object-Oriented Software Architecture:

Model-View-Controller (MVC):Three interrelated components make up the MVC architectural pattern: the Model (data and business logic), the View (user interface) and the Controller (which processes user input and updates the model). Maintainability and modularity are improved by this division. The MVC design can be used in a web application to divide the user interface (View), controller and data processing logic (Model).

Service-Oriented Architecture (SOA):Software is organised as a collection of loosely connected, self-contained services (SOAs) that communicate with one another via well-defined interfaces. Scalability and reusability are encouraged by this design. To enable communication between various components in an e-commerce system, service-oriented architecture (SOA) may be used to provide services for tasks like order processing, inventory management and payment processing.

Notes

Microservices Architecture:A system can be divided into tiny, independent services that can be created, implemented and scaled separately using the microservices architecture. Every service uses APIs to connect with other services and represents a certain business capability. Several microservices may perform tasks like file storage, picture processing and user identification in a cloud-based application, collaborating to deliver a seamless user experience.

Domain-Driven Design (DDD):Aligning software design with the business domain it represents is emphasised by the DDD approach. It entails creating a universal language (ubiquitous language) that developers and domain specialists can communicate in and structuring code according to the main ideas of the business. DDD may entail modelling terms such as “Patient,” “Doctor,” and “Appointment” in a healthcare management system according to the language used in the medical field.

5. Tools and Technologies in Object-Oriented Design:

Integrated Development Environments (IDEs):Code editors, debuggers and visual modelling features are just a few of the many tools for Object-Oriented Design that are available in IDEs like Eclipse, IntelliJ IDEA and Visual Studio. These environments facilitate the creation, modification and visualisation of OOD artefacts by developers. Developers can improve the Object-Oriented Design process in an Eclipse-based Java development environment by utilising features like real-time code analysis, code refactoring and UML diagram generation.

UML Modeling Tools:The process of creating UML diagrams is facilitated by UML modelling tools such as Lucidchart, Visual Paradigm and Enterprise Architect, which aid in visualising and documenting the behaviour and structure of software systems. Class Diagrams, Sequence Diagrams and other UML artefacts can be created and shared by team members in a collaborative software development project using a UML modelling tool.

Version Control Systems:Software project change management requires version control systems like Git, SVN and Mercurial. These solutions facilitate source code version history maintenance, tracking changes and collaboration. A version control system enables developers working on multiple facets of an Object-Oriented Design simultaneously and integrating their changes easily in a dispersed software development team.

Code Generation Tools:Code is automatically generated based on specified models or database schemas using code generation tools like CodeSmith, MyBatis Generator and Hibernate Tools. These technologies decrease the amount of manual coding required, increasing efficiency. Object-Relational Mapping (ORM) code can be generated in a database-driven application by using code generation tools, which eliminates the need for developers to write boilerplate code for data access.

6. Challenges and Considerations in Object-Oriented Design:

Overuse of Inheritance:An over-reliance on inheritance can result in unsatisfactory class hierarchies or a hierarchy that misrepresents the relationships between objects, among other design problems. It takes careful thought to weigh the advantages and disadvantages of inheritance. For example, using too much inheritance for character classes in a video game design could lead to a complicated structure that is difficult to expand.

Maintaining Consistency:It gets harder to keep design consistency when software systems change. It can take time to ensure consistency when changes are made to one area of the system and need updates to be made to several classes and components. A

modification to the interest rate calculation logic in a financial application can require revisions in several classes that handle financial computations.

Notes

Balancing Flexibility and Performance:Adding abstraction layers and interfaces is a common practice in flexible design, however it can have an adverse effect on performance. It is important to take into account finding the ideal balance between excellent performance and a flexible design, particularly in systems with strict performance requirements. It might be challenging to strike a compromise between high-performance rendering and flexible game dynamics in a real-time gaming application.

7. Real-World Applications and Case Studies:

E-Commerce Platforms:E-commerce platforms frequently use object-oriented design to model entities such as orders, products and customers. OOD's extensible and modular design enables programmers to design scalable, maintainable systems. Using Object-Oriented Design, an e-commerce programme could have classes named "Customer," "Product," and "Order," with relationships and behaviours that correspond to the business logic.

Healthcare Information Systems:

Object-Oriented Design is used in healthcare information systems to model healthcare professionals, medical procedures and patient records. Complex medical data can be represented by the application of abstraction and encapsulation. Object-Oriented Design may entail modelling classes for "Patient," "Doctor," and "MedicalProcedure," together with procedures for scheduling visits and recording medical histories, in a hospital administration system.

Content Management Systems (CMS):Object-Oriented Design is used by content management systems to model entities like authors, categories and articles. OOD's modular design facilitates the development of expandable and adaptable systems. Object-Oriented Design in a CMS could entail developing classes named "Article," "Author," and "Category," with relationships between them signifying editorial workflows and content hierarchies.

8. Future Trends in Object-Oriented Design:

Integration with Artificial Intelligence (AI):A developing trend is the combination of AI technologies and object-oriented design. AI elements can be added to object-oriented models to create more intelligent and flexible software systems. Object-Oriented Design may incorporate AI algorithms to analyse user behaviour and offer tailored content recommendations in a recommendation system.

Enhancements in Modeling Languages: Future developments in modelling languages, like UML, might bring new capabilities that improve the representation of contemporary software design ideas. This offers enhanced support for distributed systems, event-driven architectures and asynchronous programming. Future improvements to UML might include improved microservices, serverless functions and containerised deployment modelling in cloud-native applications.

Cross-Disciplinary Design:Scientific research and bioinformatics are two cross-disciplinary fields where the ideas of object-oriented design are being used more and more. The creation of software systems that simulate complicated real-world processes is the result of collaborations between domain experts from many domains and software engineers. Object-Oriented Design may be used to model biological entities and processes in a bioinformatics programme to aid researchers in the analysis of genetic data.

Notes

A fundamental and ever-evolving paradigm in software engineering, object-oriented design modelling offers a methodical way to design complex systems with an emphasis on maintainability, extensibility and modularity. This thorough overview's exploration of concepts, methods and patterns helps readers get a full understanding of Object-Oriented Design and provides developers with a road map for navigating the complexities of contemporary software development. Object-Oriented Design is still an essential framework for developing software that not only satisfies current demands but also changes with the times to accommodate emerging technologies and changing user preferences. Object-Oriented Design is still influencing software engineering today, whether it is used in content management systems, e-commerce platforms, or healthcare systems. It offers a strong basis for developing dependable, scalable and cleverly built programmes.

2.1.6 Object-Relational Design Modelling

A thorough and complex method known as "object-relational design modelling" unites the relational and object-oriented paradigms to offer a flexible foundation for creating databases that contain both data and behaviour. Through the clever integration of object-oriented concepts into the conventional relational database model, real-world entities, their relationships and the operations that can be carried out on them are represented in a more expressive and adaptable manner. This investigation will address the fundamentals, modelling strategies, language extensions and database development implications of object-relational design modelling.

A hybrid of an object-oriented and relational database model is called an object-relational model. Thus, it has support for data types, tabular structures, etc. like a relational data model and supports objects, classes, inheritance, etc. like object-oriented models. Closing the gap between relational databases and the object-oriented practices commonly employed in many programming languages, such as C++, C#, Java, etc., is one of the main objectives of the object relational data model.

Relational and object-oriented data models are both highly helpful. However, it was believed that they both lacked some qualities, thus efforts were made to create a model that combined the best aspects of each. Hence, research conducted in the 1990s led to the creation of the object relational data model.

1. Principles of Object-Relational Design Modeling:

Abstraction and Encapsulation: The abstraction principle is the foundation of object-relational design modelling and enables the representation of real-world entities as abstract data types. This is enhanced by encapsulation, which functions similarly to objects in object-oriented programming by combining behaviour and data into a single unit. For instance, in a banking system, it is possible to abstract a Customer object that contains both the customer's behaviours (such as transferFunds and makePayment) and their data (such as name and address).

User-Defined Types (UDTs):

User-Defined Types (UDTs) are a feature of object-relational databases that allow for the construction of bespoke data types that closely resemble real-world entities. Compared to conventional scalar types, UDTs offer a higher level of abstraction since they encompass both the data and related behaviours. For instance, in a logistics database, where a Location UDT contains geographic data such as latitude, longitude and distance-calculating techniques.

Notes

Inheritance and Hierarchy: Object-Relational Design Modelling embraces inheritance, a key idea in object-oriented programming. Because tables can be arranged hierarchically, offspring tables can inherit characteristics and actions from their parents. The "is-a" relationship between various entities is reflected in this hierarchy. A university database might have a hierarchy of databases with shared qualities in the Staff table and particular properties in the child tables, representing Staff, Professors and Administrators.

Complex Data Types: Beyond conventional scalar types, an extensive range of sophisticated data types are supported by object-relational databases. The modelling capabilities are improved by arrays, structures and stacked tables, which enable the representation of more intricate relationships and data structures. A video object in a multimedia database may have a nested table with comments related to the video, a structure for video information and an array of tags.

Methods and Functions: A behavioural component of Object-Relational Design Modelling is introduced by the extension of data types to include functions and methods. It is possible to define actions that can be carried out on the wrapped data by attaching methods to UDTs. Keeping with the multimedia database example, a Video UDT might include functions for interacting with users, playing back video and getting metadata.

2. Modeling Techniques in Object-Relational Design:

Entity-Relationship Modeling (ERM): Relationship between an Entity One of the fundamental methods of Object-Relational Design is still modelling. It entails recognising entities, characteristics and connections among them. Entities and tables frequently match in object-relational databases, where relationships are created via foreign keys and UDTs. Entities such as Customer, Order and Product would be recognised in an e-commerce database and relationships between them, like "Customer places Order" and "Order contains Product," would be modelled.

Class Diagrams: Class Diagrams, which are derived from object-oriented modelling, offer a visual depiction of classes, their characteristics and connections. Tables are the equivalent of classes in Object-Relational Design and associations are represented by inheritance hierarchies and foreign keys.

Classes such as Article, Author and Category would be connected in a class diagram for a content management system by associations showing relationships such as "Article written by Author" and "Article belongs to Category."

State Diagrams:The lifecycle of an entity in an object-relational database can be modelled using state diagrams, which are frequently used in object-oriented design. They show how an entity changes in response to operations and events. A state diagram in a workflow management system could show the various states an order object goes through as it moves from "Pending" to "Processing" and then "Shipped" in response to events such as order confirmation and shipment fulfilment.

Normalisation: One of the most important strategies for table structure that minimises dependencies and redundancies is normalisation. Normalisation principles are used in Object-Relational Design to guarantee data integrity while taking into account the complexity brought about by UDTs and relationships.

To prevent anomalies and duplicate information, normalisation in a healthcare database, for example, may entail organising patient data.

3. Object Query Language (OQL) and SQL Extensions:

Object Query Language (OQL): For example, an OQL query for a library database

Notes

could retrieve all books written by a particular author by navigating through the association between authors and books. Object Query Language is a query language created specifically for Object-Relational databases. It extends SQL to incorporate object-oriented concepts, allowing the expression of complex queries involving UDTs, inheritance and associations.

SQL Extensions for Object-Relational Databases:

SQL is extended by object-relational databases to support object-oriented capabilities. This involves using inheritance hierarchies, navigating relationships and running queries on UDTs. Using extensions for UDTs, a SQL query for an inventory management system might return all electronic devices along with the corresponding specs and warranty details.

4. Object-Relational Mapping (ORM):

Concept of ORM: A method called “Object-Relational Mapping” makes it easier to combine object-oriented programming languages with Object-Relational databases. It enables programmers to interact with database items in a manner that is compliant with the grammar and syntax of their chosen language. A Java object representing a Product in an ORM-enabled Java application can be mapped to a corresponding table in the database and actions on the object correlate directly to actions in the database.

Frameworks:A collection of tools and protocols for mapping objects to database tables are provided by ORM frameworks, such as Entity Framework for.NET and Hibernate for Java. A large portion of the mapping process is automated by these frameworks, which facilitates developers’ use of object-oriented paradigms to interface with databases. For instance, a Python class representing a customer can be mapped to a database table in a Python application that uses an ORM framework like SQLAlchemy and Python syntax can be used to write queries.

Pros and Cons: Benefits of ORM adoption include higher productivity, more maintainable code and a more organic alignment of application code with database structure. It does, however, have certain drawbacks, such as the requirement for meticulous optimisation, the possibility of complex mappings and performance issues.

5. Challenges and Considerations in Object-Relational Design:

Performance Considerations: Although there are many modelling options available in Object-Relational databases, there are performance issues to take into account, particularly when handling intricate queries including UDTs, associations and inheritance hierarchies. To maintain acceptable performance levels, caching methods, query optimisation and indexing become essential. When a multimedia database has UDTs for comments and video information, it becomes critical to optimise searches that use relational associations and UDT methods together.

Schema Evolution: In Object-Relational design, the way database schemas change over time can be difficult to manage. Careful migration procedures may be needed to preserve data integrity and compatibility with current applications in the event of changes to UDTs, relationships, or hierarchies. For example, in an e-commerce database, adding a new attribute to a UDT describing a Product may need schema evolution methods to update the current data.

Complexity: Adding object-oriented features can make the database schema more difficult. A careful balance must be struck between the model’s expressive potential and the simplicity needed for efficient database administration. Managing the complexity of queries

and changes becomes important when a social networking database has intricate linkages between users, posts and comments.

Notes

6. Real-World Applications and Case Studies:

Content Management Systems (CMS): When elements like articles, authors and categories are modelled using UDTs and relationships, object-relational architecture finds use in CMS databases. The expressive capacity of object-relational modelling helps CMS databases to describe intricate relationships and content structures. UDTs, for example, could be used in a CMS database to model rich content kinds, such text articles, videos and photos, with associations capturing the connections between content pieces and authors.

Geographic Information Systems (GIS): Object-Relational design is used by GIS databases to model spatial things, their characteristics and behaviours. GIS databases are essential because they contain UDTs that reflect spatial relationships, geographic locations and geographical analytic techniques. UDTs may contain data on geographical features, such as points, lines and polygons, in a GIS database, with associations reflecting the spatial relationships between the features.

Human Resources Management: HR databases use object-relational design, modelling entities such as employees, departments and roles using UDTs and relationships. Object-Relational design's adaptability makes it possible to depict intricate employee organisations like roles and hierarchies. UDTs might stand for an employee's qualifications and competencies in an HR database and affiliations could record connections between staff members and the departments to which they are assigned.

7. Future Trends in Object-Relational Design:

Integration with NoSQL Databases:

A developing trend is the assimilation of NoSQL databases with Object-Relational design concepts. This method aims to fuse the expressive power of Object-Relational modelling with the advantages of schema flexibility in NoSQL databases. When storing patient records with different features for a healthcare application, a hybrid method can use NoSQL for adaptable patient data and Object-Relational principles for intricate relationships and behaviours.

Blockchain and Smart Contracts: Concepts of object-relational design are being investigated in relation to the creation of smart contracts and blockchain. Applications built on blockchain technology can more naturally represent complex entities and interactions by utilising concepts of abstraction, encapsulation and connections. Object-Relational concepts may be used to model entities like goods, suppliers and shipments in a blockchain-based supply chain management system, along with related smart contracts to automate business logic.

Integration with Machine Learning: A new trend is the combination of machine learning models and object-relational databases. Applications can extract insights from complicated data structures by fusing the analytical power of machine learning with the modelling capabilities of Object-Relational databases.

Machine learning models could be used to evaluate consumer behaviour patterns and improve product suggestions based on links between products and customer preferences in an Object-Relational database used by an e-commerce business.

The concepts of relational databases and the adaptability and expressiveness of object-oriented modelling are smoothly combined in object-relational design modelling, which is a sophisticated and dynamic approach to database design. Object-Relational

Notes

databases continue to be a potent option for applications that demand a detailed representation of real-world things, their relationships and behaviours even as technology advances. Database designers and developers can use the concepts, modelling strategies and language extensions covered in this exploration to gain a thorough understanding of Object-Relational Design Modelling and to navigate the intricate world of contemporary data management. Object-Relational databases will continue to be crucial in determining the direction of database architecture and application development because of its embracement of the concepts of abstraction, encapsulation and relationship modelling.

2.1.7 Specialisation and Generalisation

For the purpose of creating superclass/subclass relationships, both specialisation and generalisation are helpful strategies. The following variables determine whether to apply the specialisation or generalisation strategy in a given situation:

- Nature of the problem.
- Nature of the entities and relationships.
- The personal preferences of the database designer.

Specialisation

We can state that generalisation and specialisation are mutually exclusive. To further simplify things, specialisation involves breaking things down into smaller components. Another way to put it is that in specialisation, an entity is broken down into smaller entities according to its attributes. Inheritance occurs in Specialisation as well.

Example of Specialisation

Think of an entity account. Consider the properties Acc_No and Balance for this. Additional properties like Current_Acc and Savings_Acc may be present in the account entity. Presently, Savings_Acc may have Acc_No, Balance and Interest_, whereas Current_Acc may contain Acc_No, Balance and Transactions.Calculate From now on, we can state that higher level entities' characteristics are inherited by specialised entities.

Finding subsets of an entity set (the superclass or supertype) that have a common trait is the process of specialisation. Put differently, specialisation maximises the distinctions amongst an entity's members by determining the distinctive and individual qualities (or traits) of each member. Defining superclasses and their corresponding subclasses is a top-down process known as specialisation. The superclass is usually specified first, followed by the subclasses and last by the addition of subclass-specific properties and relationship sets. Had the specialisation strategy not been used, the three subtypes would have appeared as shown in Figure (a) below.

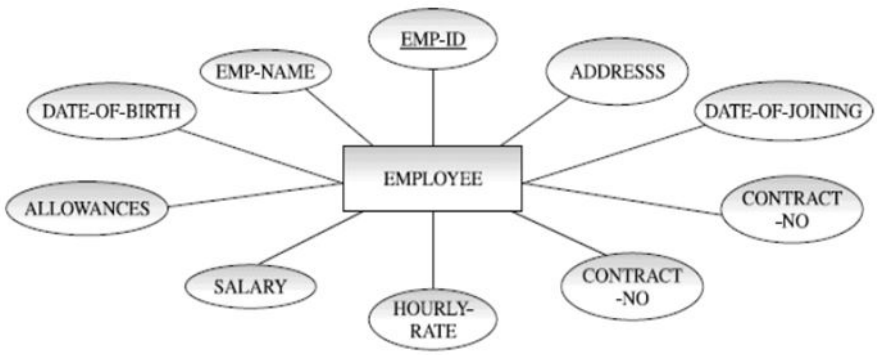


Figure: Example of specialisation

Image Source: Database Management System, S K Sinha, Second Edition

Notes

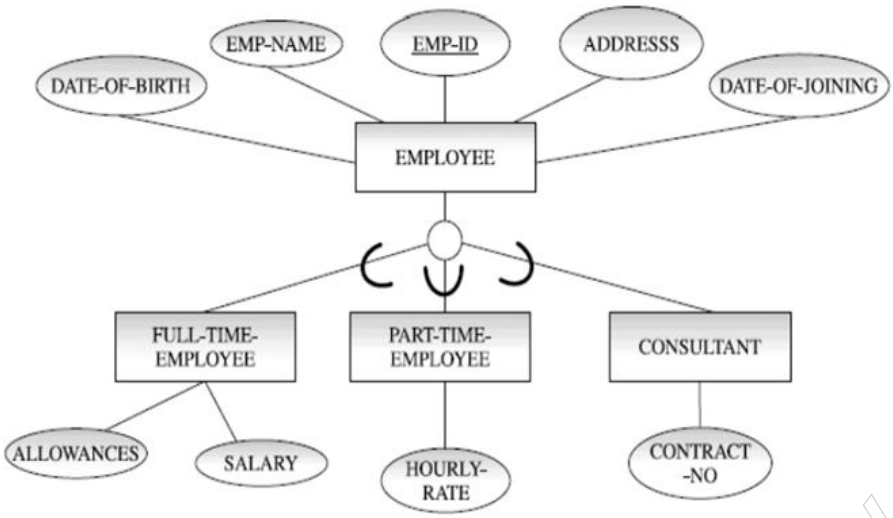


Figure: Example of specialisation

Image Source: Database Management System, S K Sinha, Second Edition

One example of specialisation is the development of three subtypes for the EMPLOYEE supertype in Figure (b) above. Numerous characteristics are shared by the three categories. However, each subtype also has certain characteristics, such as the FULL-TIME EMPLOYEE'S SALARY and ALLOWANCES. Certain subclasses also have interactions that are specific to them, such as the relationship between full-time employees and training. In this instance, specialisation has made it possible to describe the issue domain in a preferred way.

An additional illustration of the specialisation process is shown in the figure below. The entity type ITEM with the properties DESCRIPTION, ITEM-NO, UNIT-PRICE, SUPPLIER-ID, ROUTE-NO, MANUFNG-DATE, LOCATION and QTY-IN-HAND is depicted below in Figure (a). Since an item may have more than one supplier, the attribute SUPPLIER-ID is multi-valued and has the identifier ITEM-NO. Following investigation, it is now evident that the ITEM may be acquired externally or produced within. As a result, the super type ITEM can be further subdivided into PURCHASED-ITEM and MANUFACTURED-ITEM. Additionally, Figure (a) below shows that some properties are universal across all sections, independent of source. Other things, though, rely on the source. Consequently, only the MANUFACTUREDITEM subtype is covered by ROUTE-NO and MANUFNG-DATE. Likewise, UNIT-PRICE and SUPPLIER-ID are exclusive to the PURCHASED-ITEM subtype. As can be seen in Below Figure (b), PART is therefore specialised by defining the subtypes MANUFACTURED-ITEM and PURCHAED-ITEM.

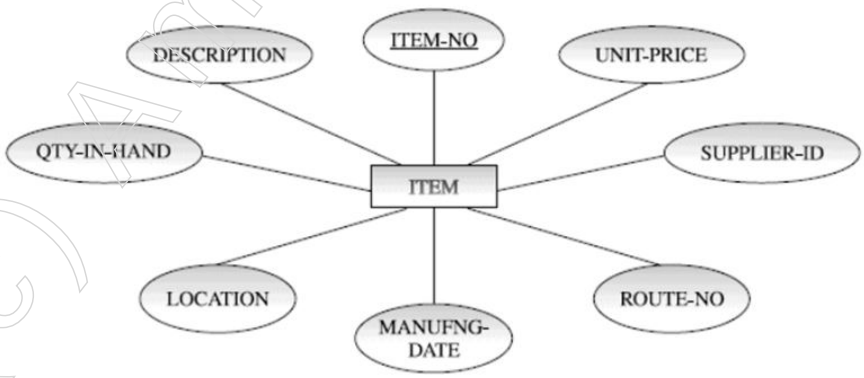
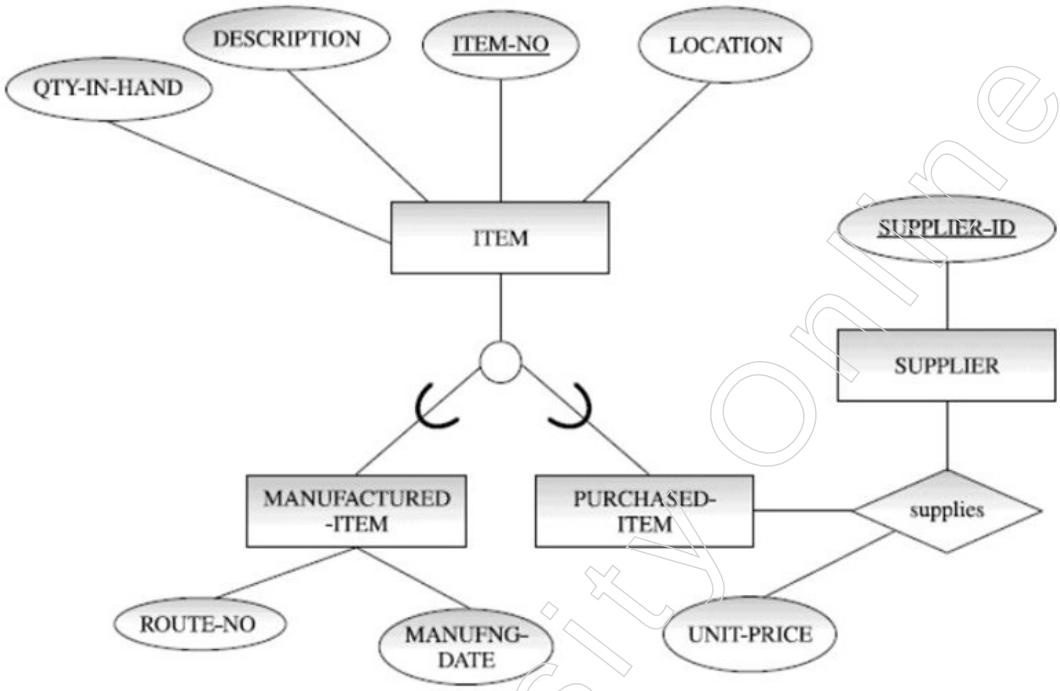


Figure: Example of specialisation

Notes



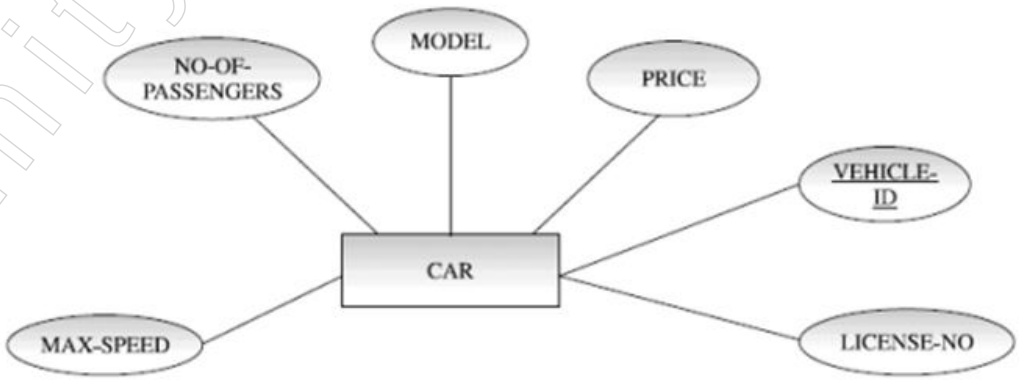
(*b*) Entity type ITEM after specialisation

Figure: Example of specialisation

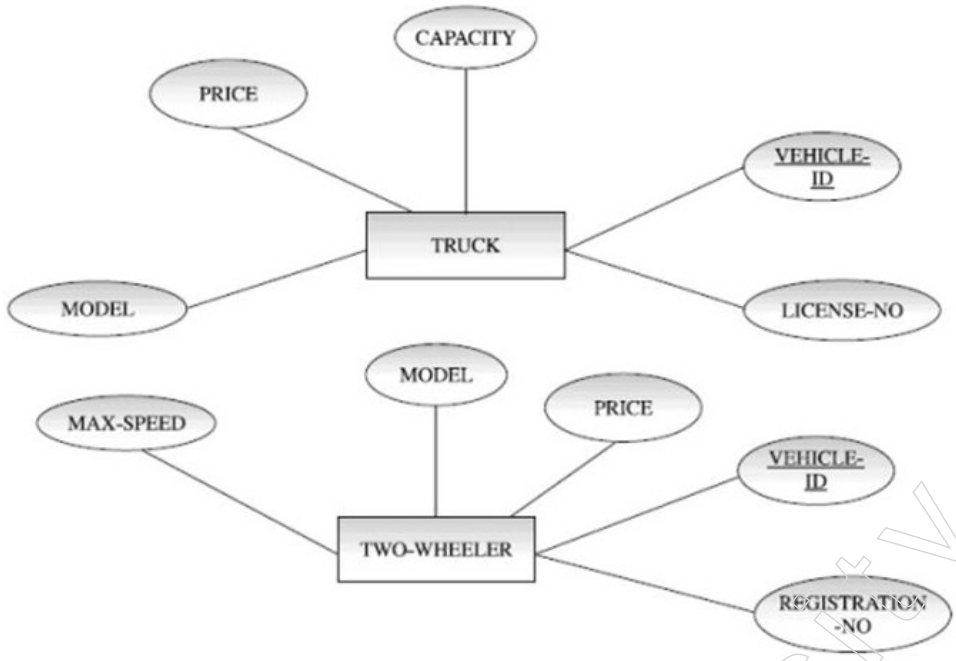
Image Source: Database Management System, S K Sinha, Second Edition

Generalisation

The practice of finding some shared traits among a group of entity sets and then building a new entity set with entities handling these shared traits is known as generalisation. Stated differently, it is the method of reducing the distinctions among the entities by recognising their shared characteristics. In contrast to specialisation, generalisation is a bottom-up process. From the initial subclasses, it denotes a generalised superclass. Usually, the superclass is declared first, followed by these subclasses and then any relationship sets involving the superclass. One example of generalisation is the creation of the EMPLOYEE superclass, which shares characteristics with three subclasses: FULL-TIME-EMPLOYEE, PART-TIMEEMPLOYEE and CONSULTANT, as seen in the figure below.

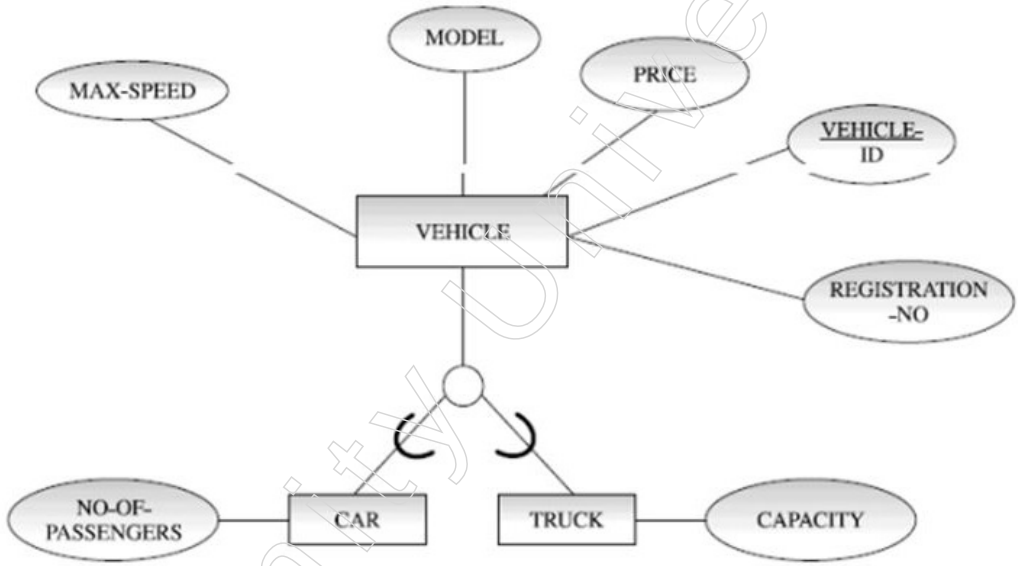


Notes



(*a*) Three entity types namely *CAR, TRUCK* and *TWO-WHEELER*

Figure: Example of generalisation



(*b*) Generalisation to *VEHICLE* type

Figure: Example of generalisation

Image Source: Database Management System, S K Sinha, Second Edition

In the accompanying Figure, another instance of generalisation is displayed. Three entity types—CAR, TRUCK and TWOWHEELER—are defined, as seen in Figure (a) above. Upon examination, it is noted that these entities share several features, including VEHICLE-ID, MODEL, PRICE, MAX-SPEED and REGISTRATION-NO. This fact essentially shows that all three of these entity kinds are variations of a common vehicle type. The generalised model of entity type VEHICLE and the ensuing supertype/subtype relationships are shown in Figure (b) above. The specific characteristic for the entity type TRUCK is CAPACITY, whereas the specific attribute for the entity CAR is NO-OF-PASSENGERS. As a result,

Notes

grouping entity types together with their shared properties has been made possible via generalisation, all the while maintaining particular attributes unique to each subtype.

It operates according to the bottom-up methodology. Lower level functions are merged in generalisation to create higher level functions, or entities. To create advanced level entities, this process is repeated one more time. Properties from specific entities are taken from them throughout the generalisation process, allowing us to build generalised entities. The process of generalisation can be summed up as follows: subclasses unite to generate superclasses.

Example of Generalisation

Think of two entities: the patient and the student. Each of these two entities will have unique qualities. For instance, the patient will have PId, Name and Mob_No features, whereas the student entity will have Roll_No, Name and Mob_No. The Generalisation Process is the ability to merge the Name and Mob_No of the Patient and Student in our example to generate a single higher level entity.

Difference between Generalisation and Specialisation:

Generalisation	Specialisation
Generalisation functions in a Bottom-Up manner.	Specialisation operates in a hierarchical manner.
The schema grows smaller as it becomes more generalised.	Specialisation results in an increase in schema size.
Usually, generalisation is used to describe a collection of items.	Specialisation is something we can do with a single thing.
One definition of generalisation is the practice of organising different entity sets into groups.	One definition of specialisation is the process of dividing an entity collection into smaller groups.
In reality, the process of generalisation involves the union of two or more lower-level entity sets in order to generate higher-level entity sets.	Generalisation is the opposite of specialisation. The process of creating a lower-level entity set from a subset of a higher-level entity set is called specialisation.
The number of entity sets is the starting point for the generalisation process, which uses some common traits to produce high-level entities.	The process of specialisation begins with a single entity set and uses several features to produce a distinct entity set.
In generalisation, a higher entity is formed by ignoring the differences and similarities between lower entities.	A higher entity splits into lower entities in specialisation.
In generalisation, inheritance does not exist.	In Specialisation, inheritance is present.

2.1.8 Aggregation and Associations

Aggregation: When two entities are seen as a single entity, this is known as aggregation. It is a relationship component in which two unique objects are involved, one of which is a component of another related instance.

Association: An association is characterised as a group of individuals with a formal structure and a shared goal. It depicts a binary relationship that characterises an action between two things. It is an object-to-object relationship. For instance, a physician may have more than one patient.

Notes

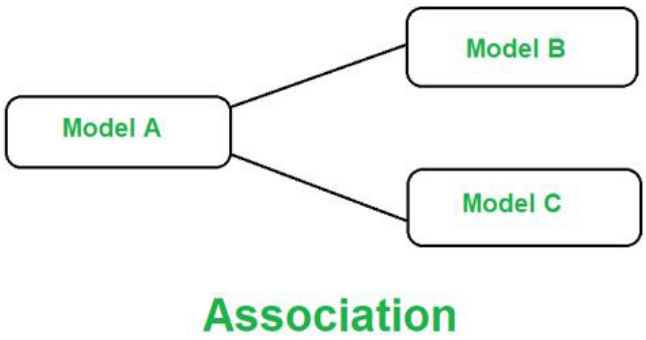


Figure: Association

Aggregation: A collection, or the getting together of items, is called an aggregate. A "has a" relationship serves as a representation of this relationship. To put it another way, an aggregate is a body, group, or mass made up of numerous unique individuals or pieces. One instance of aggregation is a phone number list.

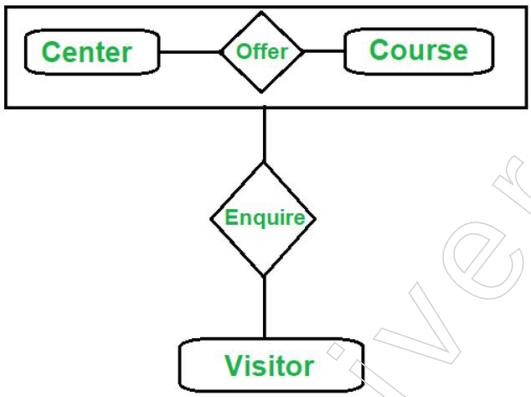


Figure: Association

Difference between Aggregation and Association:

Aggregation	Association
Aggregation is a specific kind of association that denotes a relationship between a total and a component.	When two classes are associated, it means that one class uses the other.
It is adaptable by nature.	It's rigid by nature.
unique type of interaction in which two items have a whole-part relationship	It implies that there is nearly always a connection between things.
It is shown by a relationship that "has a" + "whole-part."	A "has a" relationship serves as its representation.
The assembly class is next to a structure in the shape of a diamond.	A line segment is utilised to separate the parts or classes.

Aggregation

Organising Data:

Fundamentally, database aggregation is about arranging data into structures that make sense and are easy to use. Data in relational databases is frequently dispersed over several tables; aggregation offers a way to bring this data together, condense and show it in an organised way. As in the case of a retail establishment that tracks sales transactions

Notes

using a database. Aggregation enables the production of summary tables, such as total sales per product category or average revenue per month, in place of a single table with a large number of records. These combined tables offer a more condensed and organised perspective of the information, which facilitates analysis and insight extraction.

Aggregation Functions:

An crucial toolkit for database designers is aggregation functions. These functions work on sets of values within a column or a group of rows. They include SUM, AVG (average), MIN (minimum), MAX (maximum) and COUNT. They make it possible to compute insightful metrics and statistics that provide users insight into the properties of the data.

The total number of units sold, for example, can be obtained by using the SUM aggregation function on the "Quantity Sold" column in a sales database. Comparably, the COUNT function may be used to count the number of transactions within a given time period and the AVG function can be used to discover the average price of products.

GROUP BY Clause:

Structured Query Language (SQL)'s GROUP BY clause is a potent tool for aggregation implementation in relational databases. Users can use it to group data according to one or more columns and then individually apply aggregation functions to each group. The GROUP BY clause can be used in a customer database to group clients based on where they are located. Because of this grouping, metrics like the total number of clients, the average purchase value, or the highest number of orders from each location may be calculated by applying aggregation functions.

Rollup and Cube Operations:

Database aggregation techniques that are considered advanced include rollup and cube operations. A hierarchical perspective of the aggregated data is provided by these processes, enabling examination at various granularities.

To display total revenue at various levels of the product hierarchy, such as by category, subcategory and individual product, a rollup operation may be applied to a sales database. Cube operations go one step further by simultaneously aggregating data along many dimensions to provide a multidimensional perspective.

Hierarchical Aggregation:

Data that naturally exhibits a hierarchy or structure is a good fit for hierarchical aggregation. Data about employees' departments and divisions within departments may fall under this category.

Hierarchical aggregation, for instance, could be used in a human resources database to compile employee performance statistics at the department, division and corporate levels. This makes it possible to comprehend performance metrics at various organisational levels in a more sophisticated way.

Temporal Aggregation:

Data aggregation throughout time is the subject of temporal aggregation. Aggregation functions can be used in databases that hold temporal data, like transaction history or sensor readings, to examine trends, patterns and compile information over predetermined time periods. Calculating average stock prices or total sales on a daily, weekly, or monthly basis is one example of temporal aggregation in a financial database. Using past trends as a basis for decision-making, this temporal viewpoint facilitates the identification of patterns.

Notes

Filtering and Aggregation:

When used with filtering, aggregation enables users to concentrate on particular data subsets. Before using aggregation functions, a query's WHERE clause might have filtering rules applied to it to include or exclude particular rows. Aggregation functions, for example, could be used in a customer database to determine the average purchase amount for a specific product category, taking into account only those customers who made purchases above a predetermined threshold. More accurate insights about consumer behaviour are provided by this tailored aggregation.

Performance Considerations:

Although aggregation makes data easier to read and understand, it's important to think about the performance implications, particularly for large datasets. Resource-intensive aggregation processes may require effective indexing and query optimisation in order to keep performance levels within reasonable bounds. In order to increase the effectiveness of aggregation queries, database managers frequently employ strategies like materialised views, pre-aggregation and strategic indexing. The goal of these tactics is to reduce the amount of computing power needed to combine big datasets.

Data Warehousing:

A fundamental component of data warehousing is aggregation. Data is extracted, transformed and loaded (ETL) into a centralised repository from several source systems in a data warehousing environment. Then, aggregated views of the data that are useful for reporting and analytics are produced by applying aggregation functions. A data warehouse may hold aggregated information on sales, customer demographics and product performance in a business intelligence setting. Pre-aggregated views simplify reporting procedures, enabling quicker and more effective business metrics analysis.

Real-World Applications:

In practical applications, aggregation is ubiquitous and influences decision-making across a wide range of domains. Patient data may be combined in the healthcare industry to examine disease prevalence among various demographic groups. Aggregated traffic data can help with infrastructure development and urban planning in the transportation sector. Think of a logistics company that is in charge of a car fleet. Fuel usage, delivery times and maintenance expenses for various routes and cars might all be summed up via aggregation. This combined data helps to maximise operating efficiency, save expenses and optimise routes.

Challenges and Considerations:

Aggregation has a number of advantages, but it also has drawbacks. Finding the ideal ratio between summary and granularity might be difficult. While insufficient aggregation may result in overwhelming volumes of data that are difficult to analyse, excessive aggregation can cause a loss of detail. Furthermore, great thought must go into selecting the aggregation functions and the columns to which they are applied. Inappropriate aggregation function or column choices could cause results to be interpreted incorrectly and conclusions to be drawn incorrectly.

Emerging Trends:

As databases develop and data becomes more complicated, new aggregation tendencies are appearing. Using machine learning techniques for intelligent aggregation is one such trend. The accuracy of insights obtained from aggregated data can be enhanced

Notes

by these algorithms' ability to adaptively modify aggregation procedures in response to data trends. The introduction of serverless architectures and cloud-based databases, which provide scalable and affordable options for managing large-scale aggregation operations, is another trend. With the help of these technologies, businesses may take use of aggregation's advantages without being constrained by conventional infrastructure.

Database aggregation is a complex and essential idea that permeates all aspects of data analysis and management. Aggregation helps people extract valuable information from large datasets for use in data warehousing, relational databases and strategic planning. This helps with decision-making and optimises the system. its uses cut across many industries, demonstrating its pervasive importance in turning unprocessed data into meaningful insights. Aggregation is still a key component of databases as they develop, allowing them to adjust to new technologies and approaches while guaranteeing that the power of data is used efficiently for a variety of purposes.

Association

In the world of databases, association is a basic idea that supports the connections between the entities in a database schema. It is essential for establishing the relationships and dependencies between several tables, giving the structure required for accurately reflecting real-world circumstances. Both relational and object-oriented database models rely heavily on the idea of association, which affects how data is managed, arranged and queried. This thorough investigation will address the types, implementation, importance and effects of association in databases, as well as how it affects data modelling.

Significance of Association in Databases:

Fundamentally, association is the representation of the relationships that exist in the actual world between the things in a database. Whether simulating a consumer making purchases in an online store or workers in designated departments inside a company database, affiliations represent the complex network of relationships that characterise the integrity and functionality of the system. Imagine a database used for online shopping where clients can place product orders. In order to track which customer placed which order and create a relational relationship that emulates the interaction between customers and orders in the real world, the "Customer" and "Order" entities must be associated.

Types of Association:

There are several sorts of associations in databases, each of which represents a particular form of link between elements. The principal varieties consist of:

One-to-One (1:1) Association:Every record in one table is linked to exactly one record in another table in a one-to-one association and vice versa. Although it's not very common, this kind of association is used when entities need their qualities to be separated despite their close relationship.

For instance, a one-to-one linkage could be used to associate each employee with a specific parking spot in a database that tracks employee data.

One-to-Many (1:N) Association:One-to-many relationships are more prevalent and indicate that while each record in the second table is related with just one record in the first table, each record in the first table can be associated with several records in the second table. As in the case of a university database where the "Professor" and "Department" tables are associated one to many times. Professors can work in more than one department, but they are all affiliated with just one department.

Notes

Many-to-Many (M:N) Association:Each record in one table may have many associations with records in another table and vice versa, according to many-to-many associations. To resolve the many-to-many relationship in this kind of association, a junction or associative table must be introduced.

It is possible to create a many-to-many association between the “Book” and “Author” tables in a database for a library system. The various writers connected to each book would be included in a junction table and vice versa.

Implementation of Association:

Foreign keys and referential integrity restrictions are used in databases to implement associations. A field in one table that creates a connection between it and another table is called a foreign key. The primary key of another table is cited in the creation of this link, confirming the relationship between the two entities.

Foreign Keys:Using foreign keys is essential for putting associations into practice. A foreign key that references the primary key of the table on the “one” side is usually present in the “many” side of a one-to-many association.

To create the relationship between songs and albums, a database representing a music library might, for example, contain a foreign key in the “Song” table that references the primary key in the “Album” table.

Referential Integrity Constraints:

Constraints on referential integrity guarantee the validity of table relationships. These limitations stop activities that lead to references to nonexistent records or orphaned records. bReferential integrity constraints would make sure, to stick with the music library example, that a song could never be connected to an album that didn’t exist in the “Album” database.

Impact on Data Modeling:The way data is modelled in a database is greatly influenced by the efficient usage of associations. It makes accurate and flexible modelling of complicated real-world circumstances possible. Identifying entities, specifying their characteristics and creating associations that represent the connections between these entities are all steps in the modelling process.

Entity-Relationship Diagrams (ERDs):A visual depiction of the database schema that shows entities, characteristics and the relationships between them is called an entity-relationship diagram (ERD). These diagrams give a clear visual representation of the relationships and act as a template for database architecture. Relationships between related entities are depicted by lines in an Entity Relationship Diagram (ERD). These lines are frequently marked with cardinality indications to indicate the type of relationship (e.g., one-to-many, many-to-many).

Normalisation:Tables are arranged as part of the normalisation process, an essential stage in database architecture, to lessen dependencies and redundancies. Associations are essential to the normalisation process because they direct table decomposition in a way that protects data integrity and gets rid of abnormalities.Think of a database that records client orders. The normalisation process is influenced by the relationship between orders and customers, which guarantees that data is efficiently arranged without needless duplication.

Impact on Querying and Retrieval:

Associations have a significant influence on database querying and retrieval. Navigating associations enables the answer of challenging questions and the extraction of significant discoveries.

Notes

JOIN Operations:In order to query databases containing associations, JOIN operations are essential. They make it possible to combine data from different tables according to the designated associations. Data can be extracted while maintaining the associations' defined linkages thanks to the INNER JOIN, LEFT JOIN and RIGHT JOIN operations. A list of authors and their corresponding books can be retrieved via a JOIN operation in a database where "Author" and "Book" have a one-to-many link.

Subqueries:Subqueries are frequently employed in more complex inquiries to take advantage of associations. Subqueries facilitate the execution of nested inquiries inside the primary query, allowing information to be retrieved based on associations with additional data sets. Using the relationship between customers and orders, a subquery in a sales database may be used to identify customers who made purchases over a specific threshold.

Aggregation and Grouping:Associations are involved in the processes of grouping and aggregation as well. Associations specify how records are grouped together when aggregating data according to specific criteria, impacting the results of aggregation operations.

Aggregation may, for instance, use the relationship between books and branches to get the total number of books checked out by each library branch in a database for a library system.

Challenges and Considerations:Although associations are necessary for precise and thorough database modelling, there are certain issues with them that database designers need to deal with.

Performance Considerations:Performance issues can arise from associations, specifically many-to-many relationships, especially when working with big datasets. To mitigate performance issues, effective database design, query optimisation and index use are essential.

Complexity and Maintenance:Managing associations can get difficult as databases get more complex. Database schema updates may be required due to new entity additions, data relationship changes and evolving business requirements. Database administrators need to think carefully about how these modifications may affect current associations and the integrity of the system as a whole.

Balancing Normalisation and Denormalisation:In database design, finding the ideal balance between normalisation and denormalisation is a continuous process. Denormalisation could be required to maximise query performance, even if normalisation seeks to remove duplication and guarantee data integrity. It is important to carefully evaluate these factors while making decisions on associations and schema design.

Real-world Applications:Associations have applications in a wide range of fields and sectors, which affects how databases are created to satisfy certain business requirements.

Healthcare Systems:Relationships between patients, medical records, therapies and healthcare providers are captured by associations in healthcare databases. This makes it possible to have a comprehensive understanding of patient care, giving medical personnel access to pertinent data so they may decide with knowledge.

Supply Chain Management:In supply chain databases, associations play a crucial role since items like orders, products and suppliers are closely connected. Having clearly defined associations in the database architecture is essential for tracking the movement of items, controlling inventory and streamlining logistics.

Social Media Platforms:In order to model relationships between users, posts, comments and other activities, social media systems mostly rely on associations. These

connections make it easier to display content in a personalised manner, suggest friends and display ads that are specifically tailored to the actions of the user.

Notes

Financial Systems:Associations are used by financial databases to show the connections between clients, accounts, deals and assets. This makes it possible to generate financial reports, accurately track financial activity and comply with regulatory obligations.

Future Trends and Technologies:Emerging trends and technologies that affect the modelling and management of associations are hallmarks of the changing database landscape.

Graph Databases:The emergence of graph databases brings with it a new way of modelling and querying associations. Graph databases are ideal for situations where linkages are heavily interwoven, such social networks, fraud detection and network research. They are excellent at expressing and navigating complex relationships.

NoSQL Databases:NoSQL databases offer an alternative to conventional relational databases because of their distributed architecture and customisable structure. These databases work well in situations where associations might change quickly, giving users more flexibility to adjust to shifting business needs.

Blockchain and Distributed Ledgers:The technologies of distributed ledgers and blockchain present new models for decentralised association management. These technologies are appropriate for applications like supply chain traceability and secure transaction management because they offer clear and unchangeable records of associations.

The fundamental idea of association in databases influences the design and operation of contemporary data management systems. Assigning values have a significant impact on query performance, system integrity and modelling intricate real-world relationships. They are also essential to the design and development of databases. Understanding and making good use of associations are still essential for creating databases that faithfully capture the complexities of the systems they reflect, even as technology advances. Associations are the threads that bind data together in standard relational databases, NoSQL databases, or specialised graph databases, allowing organisations to extract insights and take well-informed decisions.

2.1.9 Object Query Language

The query language suggested for the ODMG object model is called object query language, or OQL. It is intended to function closely with programming languages like Java, Smalltalk and C++ for which an ODMG binding is defined. As a result, objects that fit the type system of the programming language in question can be returned by an OQL query contained within it. Additionally, these programming languages can be used to write the code for the class operations implementations in an ODMG schema. Similar to the relational standard query language SQL, OQL has additional functionality for ODMG concepts including inheritance, polymorphism, relationships, complex objects, operations and object identity.

A standardised query language called Object Query Language (OQL) was created expressly for use with object-oriented databases (OODBs). It acts as a link between the object-oriented paradigm and the database query specifications. Users can access, modify and traverse intricate data structures kept in object-oriented databases with the aid of OQL.

Notes

Key Concepts of OQL:

Object-Oriented Model:

- The foundation of Object-oriented QL is the object-oriented data model;
- objects in OQL represent both data and behaviour and
- interactions between objects are represented by means of notions such as inheritance, aggregation and associations.

Querying Objects:

- With OQL, users may create queries that retrieve objects according to predefined standards.
- Filtering, sorting and projecting data from database-stored objects are all possible with queries.

Navigating Relationships:

- OQL's aptitude for navigating intricate interactions between objects is one of its advantages.
- It is easy for users to navigate associations and find related objects.

Declarative Syntax:

- Because OQL has a declarative syntax, users can define the data they want without having to specify how to get it.
- In imperative languages, on the other hand, the emphasis is on defining the steps necessary to accomplish a goal.

Structured Query Language (SQL) vs. OQL:

- OQL and SQL, the common query language for relational databases, are frequently contrasted.
- OQL is suited to the hierarchical and nested structures of object-oriented databases, whereas SQL is meant for tabular data.

OQL and Unified Modeling Language (UML):

- The Unified Modelling Language (UML), which is frequently used in object-oriented design and modelling and OQL frequently coincide.
- UML diagrams, like class diagrams, can be converted into OQL queries to obtain particular object instances.

Basic OQL Syntax:

SELECT Statement: The properties or attributes of the objects to be obtained are specified using the SELECT command.

Example: SELECT name, age FROM Person

FROM Clause:The object types or classes from which data is to be fetched are specified in the FROM clause.

Example: FROM Person

WHERE Clause: The WHERE clause applies filters to objects according to predefined criteria.

Notes

Example: WHERE age > 21

ORDER BY Clause: The result set is sorted using the ORDER BY clause according to predefined attributes.

Example: ORDER BY name ASC

JOIN and Navigation: Dot notation in OQL enables the navigation of relationships between items.

Example: FROM Company c, c.employees e WHERE e.age > 30

Advanced OQL Concepts:

Aggregation Functions: Aggregation functions such as SUM, AVG, MIN and MAX can be used in OQL to conduct computations on grouped data.

Example: SELECT AVG(salary) FROM Employee

Grouping: To aggregate query results according to certain attributes, use the aggregate BY clause.

Example: SELECT department, AVG(salary) FROM Employee GROUP BY department

Subqueries: Subqueries are supported by OQL, enabling the use of one query's results in another.

Example: SELECT name FROM Employee WHERE department IN (SELECT name FROM Department WHERE location='New York')

Polymorphism: Because OQL manages polymorphism, queries can return objects from both base classes and their subclasses.

Example: SELECT * FROM Shape

Indexing and Optimisation: Indexing techniques can greatly enhance OQL queries by improving data retrieval. Optimising query performance, particularly in large databases, requires indexing.

Use Cases and Applications:

Database Retrieval: Object-oriented databases can be queried using OQL to get objects and their characteristics. It is possible to customise queries to reflect the intricate architecture of objects.

Object-Oriented:Object-oriented models, like UML diagrams, can be more easily translated into executable queries with the help of OQL. In object-oriented systems, this aids in bridging the gap between design and implementation.

Complex Relationships: OQL is a powerful query language for databases with complex object relationships. An important strength is navigating associations and gaining access to connected things.

Data Analysis: OQL can be used for data analysis jobs when specific querying is necessary due to the layered and hierarchical nature of data structures.

Challenges and Considerations:

Performance Optimisation: The optimisation of OQL query performance necessitates careful consideration of query structure and indexing, particularly in big databases.

Standardisation: OQL does not have a common syntax throughout OODBMS implementations. Different systems may have different syntax.

Notes

Learning Curve: Learning OQL has a learning curve, much like learning any other query language, especially for people who are used to relational database query languages like SQL.

Object-oriented databases can be queried with great effectiveness using Object Query Language (OQL). It is an invaluable tool in the creation and upkeep of object-oriented systems because of its declarative syntax, ability to handle intricate interactions and conformity to the principles of object-oriented modelling. Even though there are still obstacles, developments in OQL and its interaction with object-oriented databases let modern software developers retrieve data more effectively and expressively.

2.1.10 Object-Relational Concepts

By mapping an object's state to a database column, Object Relational Mapping (ORM) is a feature that helps create and preserve a relationship between an object and a relational database. It can readily perform a variety of database tasks, including inserting, updating and deleting data.

The integration of object-oriented programming concepts into relational database management systems (RDBMS) is known as object-relational notions. By combining the advantages of relational and object-oriented modelling, this hybrid approach seeks to enable more expressive and flexible data modelling. The following are important ideas related to object-relational databases:

Object-Relational Model: The relational model's components and the object-oriented paradigm's features are combined to create the object-relational model.

Data Abstraction: It gives data modelling a higher level of abstraction and makes it possible to represent real-world things with characteristics and actions.

User-Defined Types (UDTs): User-Defined Types (UDTs) are data types that users can define in object-relational databases.

Encapsulation: UDTs are similar to objects in object-oriented programming in that they include both data and behaviour.

Inheritance: A key idea in object-oriented programming is inheritance, which allows a new class to take on properties and functions from an older class. Tables can be arranged hierarchically in object-relational databases, with child tables inheriting properties from parent tables.

Complex Data Types: In addition to typical scalar kinds, object-relational databases offer complicated data types like arrays, structures and user-defined types. This makes it possible to depict intricate relationships and data structures more accurately.

Methods and Functions: User-defined types can have methods or functions attached to them in object-relational databases. By introducing behavioural elements to data types, this makes it possible to define the operations that can be carried out on the data.

Object Identity: Every object or row in an object-relational database has a unique identifier that helps to set it apart from other objects. Relationship maintenance and data integrity depend on object identity.

Relationships and Joins: Table relationships are shown and joins may entail intricate structures like arrays and nested tables. This enables for the representation of more complex relationships and improves the expressiveness of queries.

Query Language Enhancements: SQL Extensions: Object-oriented capabilities in Object-Relational databases are supported by extensions to SQL (Structured Query

Language). Complex data structures may be navigated and queried thanks to improved query capabilities.

Notes

Object-Relational Mapping (ORM):

An object-oriented language's objects can be mapped to relational database tables using the ORM approach. The smooth integration of object-oriented code with an object-relational database is made possible by ORM frameworks.

Indexes and Performance: The purpose of indexing techniques is to maximise query performance when dealing with complicated data types. Indexes are essential for increasing the retrieval speed of complex queries.

Transaction Management:ACID qualities: To guarantee transactional integrity, object-relational databases, like conventional relational databases, conform to the ACID qualities (Atomicity, Consistency, Isolation, Durability).

Security:In object-relational databases, security measures are put in place to manage data access. User Authorisation and Authentication: These features guarantee that data can only be accessed and modified by authorised users.

Concurrency Control:Mechanisms for controlling concurrent access to data by numerous users or programmes are included in object-relational databases. Concurrent transactions are handled by versioning and locking strategies.

Integration with Programming Languages:Object-oriented programming languages and object-relational databases are intended to work together harmoniously. Developers can interact with database objects in a programming language-consistent manner thanks to object-relational binding.

Mapping Directions

There are two sections to the mapping directions: -

- Unidirectional relationship - Only one entity in this connection has the ability to refer to another's properties. It simply has one due side, which outlines the procedure for doing a database update.
- Bidirectional relationship - Both an inverse and an owning side exist in this connection. There is a relationship field or property reference for each entity in this instance.

Types of Mapping

The different ORM mappings are as follows: -

- One-to-one: The @OneToOne annotation represents this association. In this case, one instance of one entity is connected to another instance of that other entity.
- One-to-many: The @OneToMany annotation represents this association. An instance of one entity may be associated to several instances of another entity in this relationship.
- Many to one The annotation @ManyToOne defines this mapping. Multiple instances of one entity can be associated to a single instance of another entity under this connection.
- Many-to-many: The @ManyToMany annotation represents this association. In this case, several instances of one entity may be connected to several instances of another. Any side may be the owing side in this mapping.

Combining the rich flexibility of object-oriented programming with the usually inflexible structure of relational databases, the combination of Object-Relational concepts denotes a sophisticated approach to database management. The goal of this combination is to

Notes

create a more complete and expressive model for expressing complex data structures by facilitating a smooth transition between the inheritance, polymorphism and encapsulation of the object-oriented paradigm and the tabular structure of the relational model.

The foundation of Object-Relational notions is the introduction of User-Defined Types (UDTs), which allow data and related behaviours to be combined into a single entity. By doing this, the emphasis is shifted from simple data storage to the representation of actual entities as complete units, much like object-oriented programming's objects. These UDTs can represent a wide range of data types, including more complex structures like arrays and structures in addition to the more common scalar kinds, enabling modelling at a greater level of detail.

A key idea in object-oriented programming, inheritance is used in object-relational databases to provide a hierarchical table organisation method. This design creates a distinct "is-a" relationship between various entities by having child tables inherit attributes and functions from their parent tables. This hierarchical structure makes it easier to organise data so that it more closely resembles the relationships that exist naturally between different entities in the actual world, which improves conceptual clarity and maintainability.

To close the gap between data and behaviour, object-relational databases additionally present the idea of UDT-related methods and functions. This extension gives the database schema an extra layer of capability by allowing the definition of actions that can be carried out on the contained data. As a result, entities are represented more cogently, with a single structure encapsulating both their ability to perform actions and store data.

Object-Relational databases maintain object identity, which is essential to object-oriented programming. Every object or row in a database has an identity that is specific to it, so relationships are kept accurate. Maintaining the integrity of the data and navigating the intricate associations found in the database are made possible by this unique identity.

To handle the intricacies of the object-oriented model, object-relational databases expand the capabilities of Structured Query Language (SQL) in the field of querying. With the ability to navigate complex relationships between items, queries can now get data in a more expressive and natural way. The flexibility and strength of searches are increased and they become more in line with the intricacies of real-world data structures, thanks to the ability to effortlessly browse associations and retrieve related objects.

The capabilities of the Object-Relational paradigm extend beyond querying to include indexing and performance optimisation. Complex data types have unique characteristics that indexing algorithms are designed to manage, resulting in the efficient execution of queries using these types. Transaction management in Object-Relational databases follows the fundamental principles of ACID (Atomicity, Consistency, Isolation, Durability), which becomes especially important in scenarios where large datasets and complex relationships require sophisticated optimisation strategies to maintain acceptable performance levels. These guidelines guarantee the consistent execution of transactions, preserving data integrity even when numerous users or programmes are attempting simultaneous access. To manage the difficulties presented by concurrent transactions, strategies like locking and versioning are used, guaranteeing a stable and reliable transactional system.

Another essential component weaved into the Object-Relational framework is security. To manage access to data, user authentication and authorisation mechanisms are put in place. This protects sensitive data from unauthorised access by guaranteeing that only authorised users can interact with and edit the data.

Notes

Moreover, object-oriented programming languages and object-relational databases combine smoothly to create a cohesive development environment. This integration expedites the development process and promotes code reuse by enabling developers to work with database objects in a way that is consistent with the programming language they are using.

Essentially, the notions of Object-Relational mark a paradigm change in the organisation and use of databases. These ideas open the door to a more scalable, expressive and intuitive method of data modelling and retrieval by fusing the benefits of object-oriented principles with the relational model's strengths. The databases that are produced are dynamic entities that capture the structure and behaviour of real-world things, offering a more accurate representation of the subtleties of the systems they represent than just stores of data.

Summary

- An Object-Oriented Database (OODB) is a type of database management system (DBMS) that is designed to store, manage and retrieve complex data structures modeled after real-world entities using the principles of object-oriented programming. Unlike traditional relational databases, which use tables and rows to organise data, object-oriented databases use objects, classes and inheritance to represent and store information. Object-Oriented Databases are particularly suitable for applications with complex data structures, where the relationships and behaviors of objects closely mirror real-world entities. They offer a natural way to model and manage data in object-oriented programming paradigms.
- Object-oriented (OO) concepts are fundamental principles and practices in object-oriented programming (OOP), a programming paradigm that uses objects, classes and related concepts to structure and organise code. These concepts form the foundation of object-oriented programming and provide a powerful way to model and structure code, making it more modular, reusable and easier to understand and maintain.
- The architecture of an Object-Relational Database Management System (ORDBMS) combines elements of both traditional relational databases and object-oriented databases to handle complex data structures. ORDBMS integrates object-oriented features into the relational database model, allowing for the storage and retrieval of more complex data types, such as objects, arrays and user-defined types. Object-Relational Database Management Systems provide a bridge between the relational and object-oriented paradigms, offering the benefits of both worlds and accommodating the storage and retrieval of diverse and complex data structures.
- The architecture of an Object-Oriented Database Management System (OODBMS) is designed to support the storage, retrieval and management of complex data structures based on object-oriented principles. Unlike traditional relational databases, OODBMS treats data as objects, where each object encapsulates both data and methods. Object-Oriented Database Management Systems are particularly suitable for applications where the data model closely mirrors real-world entities and where complex relationships and behaviors need to be represented and managed. They provide a seamless integration of object-oriented programming principles into the database environment.
- Object-Oriented Design (OOD) is a key aspect of the software development process that focuses on creating a system's architecture and design using object-oriented principles. Object-Oriented Design Modeling involves the creation of visual representations and diagrams to illustrate the structure, behavior and relationships of the objects within a system. Object-Oriented Design Modeling using UML and associated diagrams is

Notes

crucial for communicating and visualising the system's architecture and behavior during the design phase of software development. These diagrams aid in understanding, documenting and refining the system design before moving to the implementation stage.

- Specialisation and generalisation are two fundamental concepts in Object-Oriented Design that are used to model relationships and hierarchies between classes. These concepts are often associated with inheritance and help create a more organised and flexible class structure. Specialisation, also known as inheritance or subclassing, is a process where a new class (subclass or derived class) is created by inheriting attributes and behaviors from an existing class (superclass or base class).Generalisation is the process of creating a more general or abstract class from a set of more specialised classes. It involves identifying common attributes and behaviors shared by multiple classes and creating a more generalised class. These concepts play a crucial role in creating hierarchies and organising classes in object-oriented systems, providing a means to represent both the commonalities and differences among classes.
- Aggregation and association are both concepts in object-oriented design that define relationships between classes, but they represent different kinds of relationships with varying levels of dependency. Association represents a more general relationship between classes. It signifies that objects of one class are related to objects of another class in some way, but it does not imply a specific type of relationship. Aggregation is a type of association that represents a "whole-part" relationship between a whole object and its parts. It signifies that one object (the whole) is composed of or has components that are themselves objects (the parts), while both aggregation and association represent relationships between classes, aggregation specifically denotes a "whole-part" relationship where the parts can exist independently, while association is a more general term indicating a connection between classes. The choice between them depends on the nature of the relationship you want to model in your system.
- Object Query Language (OQL) is a query language used for querying object-oriented databases. It is designed to work with data models that are based on the principles of object-oriented programming, where data is stored in the form of objects. OQL allows users to express queries to retrieve, manipulate and update objects stored in an object-oriented database. While OQL has been used in some object-oriented database systems, it's important to note that the adoption of OQL has not been as widespread as SQL for relational databases. Many modern object-oriented systems may use alternative approaches or query languages.
- Object-Relational Concepts refer to the integration of object-oriented programming principles into the relational database model. Object-Relational Database Management Systems (ORDBMS) aim to bridge the gap between the object-oriented paradigm, commonly used in application development and the relational model, which has been a standard for database management. Object-Relational concepts aim to provide a more seamless integration of object-oriented programming and relational databases, allowing developers to work with a data model that closely aligns with the structure of their application code. These concepts enhance the expressiveness and flexibility of database systems, supporting the development of more robust and maintainable applications.

Glossary

- SQL: Structured Query Language
- DBMS: Database Management Systems
- OODBs: Object-Oriented Databases

Notes

- GUIs: Graphical User Interfaces
- OODBMSs: Object-Oriented Database Management Systems
- SDM: Semantic Data Model
- OODM: Object-Oriented Data Model
- ORDM: Object-Relational Data Model
- ERDM: Extended-Relational Data Model
- OOPLs: Object-Oriented Programming Languages
- OID: Object Identifier
- CDM: Conceptual Data Modelling
- E-R: Entity-Relationship
- OO: Object-Oriented
- DMLs: Data Manipulation Languages
- CAD: Computer-Aided Design
- OIS: Office Information Systems
- CASE: Computer-Aided Software Engineering
- UDTs: User-Defined Types
- APIs: Application Programming Interfaces

Check Your Understanding

1. What is an Object-Oriented Database (OODB)?
   a) A type of relational database
   b) A database that uses object-oriented principles for data representation and management
   c) A database that exclusively stores objects and no other data types
   d) A database used only for object-oriented programming languages
2. In an Object-Oriented Database, what is an Object?
   a) A record in a table
   b) A unique identifier for a database entry
   c) An instance of a class with attributes and methods
   d) A relationship between two tables
3. What is Inheritance in Object-Oriented Programming?
   a) A process of creating objects from classes
   b) A mechanism for code reuse and creating a hierarchy of classes
   c) The process of defining the structure of a class
   d) A way to represent relationships between objects
4. What is Encapsulation in Object-Oriented Programming?
   a) The process of creating a copy of an object
   b) The bundling of data and methods that operate on the data within a class
   c) A mechanism for accessing private data outside the class
   d) The process of hiding the implementation details of a method

Notes

5. What is the purpose of Object-Relational Mapping (ORM) in an ORDBMS?
   a) To create a visual representation of database tables
   b) To facilitate communication between the application and the database
   c) To map objects to relational tables and vice versa
   d) To enforce security measures in the database

Exercise

1. Explain in detail object-oriented database and OO concepts.
2. Explain briefly architecture of object-relational database management system and architecture of object-oriented database management system.
3. What is object-oriented design modelling and object-relational design modelling?
4. Differentiate between specialisation and generalisation. Explain with examples.
5. What is the difference between aggregation and associations?

Learning Activities

1. Explain the key principles and features of Object-Oriented Databases (OODB). Discuss how OODB differs from traditional relational databases and highlight situations where OODBs are particularly advantageous. Provide examples to illustrate your points.
2. Discuss the principles and techniques involved in Object-Oriented Design (OOD) modeling. Explain how OOD contributes to the development of robust and maintainable software systems. Provide examples to illustrate the application of key modeling concepts.

Check Your Understanding- Answers

1. b 2. c 3. b 4. b
5. c

Module -III: Parallel and Distributed Database

Notes

Learning Objectives

At the end of this module, you will be able to:

- Understand Parallel Database
- Analyse Distributed Databases Principles
- Define Architectures Of Parallel Databases
- Define Transparencies In Distributed Databases
- Analyse Query Processing In Distributed Database

Introduction

Two different but related database paradigms—parallel and distributed databases—were created to tackle the difficulties of efficiently handling enormous volumes of data. A parallel database enables the simultaneous execution of queries and transactions by distributing data processing among several processors operating within a single system. Through the use of several processors' combined computational capacity, this parallelism improves performance.

Distributed databases, on the other hand, expand on this idea by distributing data over a network of linked computers, or nodes. The system as a whole gains from fault tolerance, scalability and parallel processing as each node runs independently. These databases can manage massive datasets and users who are spread out globally because of its distributed architecture.

The goals of distributed and parallel databases are to overcome the drawbacks of conventional centralised databases and enhance performance. While distributed databases offer scalability and fault tolerance by distributing data and processing across a network of connected machines, parallel databases perform best in situations when a single powerful machine with several processors can be used efficiently. The scale of the dataset, the geographic dispersion of users and the particular needs of the application at hand are some of the variables that influence the decision between these paradigms.

3.1 Advanced Database Systems

The underlying computer system that a database system operates on has a significant impact on the architecture of the database system. Systems for databases might be distributed, parallel, or centralised. Every piece of information is kept up to date on a single location (or computer system) and it is presumed that each transaction is processed effectively in sequence. However, modern databases are required to handle more complex and demanding requirements from users, such as high performance, increased availability, distributed data access, distributed data analysis and so forth and a single CPU-based computer architecture is not sufficient for them.

The architecture of today's advanced database systems uses multiple CPUs operating in parallel to deliver complicated database services in order to meet the sophisticated expectations of users. Certain architectures involve several CPUs operating in parallel, all of which are physically close to one another within the same building and have extremely fast communication. Parallel databases are those that function in these kinds of environments.

Notes

3.1.1 Introduction to Parallel Database

Parallel databases, which distribute and process data across numerous processors or nodes simultaneously, are a significant improvement in database administration. They are intended to improve performance and scalability. This method of parallel processing makes it possible to execute queries and transactions more quickly, which makes it very useful for managing heavy workloads and massive data volumes. Understanding parallel databases' fundamental ideas, benefits, drawbacks and use cases is necessary for an introduction.

A parallel database system loads and builds indexes, evaluates queries and performs other activities in parallel in an effort to increase performance. In a system like this, data can be distributedly stored, but the distribution is determined only by performance factors.

Multiple CPUs do different tasks in parallel in parallel database systems, like loading data, creating indexes and analysing queries, to increase performance. A large task can be divided into numerous smaller tasks by parallel processing, which then carries out the smaller tasks concurrently on multiple nodes (CPUs). Consequently, the bigger jobs get done faster. Database systems that use many CPUs and discs operating in parallel increase processing and input/output (I/O) speeds. Applications that have to query big databases and handle a lot of transactions per second will find parallel databases quite helpful. Unlike centralised processing, which involves serial calculation, parallel processing allows for the simultaneous execution of numerous processes.

Therefore, maintaining the database system's ability to operate at a reasonable pace despite growing database size and transaction volume is typically the aim of parallel database systems. An organisation can grow more smoothly by improving the system's capacity through increased parallelism than by switching to a faster machine to replace a centralised system.

Parallel database systems often have a unified site machine (computer) architecture and are built from the ground up to offer the optimum cost-performance. Typically, a database system's transaction management module is where site computers collaborate with one another. The goal of parallel database systems is to use multiple tiny CPUs to build a speedier central computer. Having multiple smaller CPUs that add up to the power of one larger CPU is more cost-effective.

A parallel database system loads and builds indexes, evaluates queries and performs other activities in parallel in an effort to increase performance. In a system like this, data can be distributedly stored, but the distribution is determined only by performance factors.

Parallel systems use many computers running in parallel to increase processing and input/output speeds. The study of parallel database systems is becoming more and more essential as parallel processors become more ubiquitous. In contrast to serial processing, which carries out the computing steps one after the other, parallel processing allows for the simultaneous execution of numerous processes. A massively parallel or fine-grain parallel machine uses thousands of smaller processors, whereas a coarse-grain parallel system uses a small number of powerful processors. Nowadays, almost all high-end server systems contain up to two or four processors, each with 20–40 cores, providing some coarse-grain parallelism.

The significantly higher degree of parallelism that massively parallel computers support sets them apart from coarse-grain parallel computers. Sharing memory amongst numerous CPUs is impractical. As a result, many computers—each with its own memory and frequently its own set of disks—are usually used to build massively parallel computers. In the system, each of these computers is referred to as a node. A data centre is a facility that holds a large number of servers and is used to house parallel systems at scales of

Notes

hundreds to thousands of nodes or more. High-speed network connectivity is offered by data centres both internally and outside. Over the past ten years, data centre sizes and numbers have increased dramatically; contemporary data centres may include several hundred thousand servers.

Even some of the most ardent supporters of parallel database systems had all but written them off more than 20 years ago. Today, nearly every database-system provider successfully markets them. Several tendencies drove this change:

- With the growing use of computers, organisations' transaction needs have expanded. Furthermore, the expansion of the World Wide Web has led to the creation of several websites with millions of visitors and the growing volume of data gathered from these visitors has resulted in extraordinarily enormous databases at numerous businesses.
- Businesses utilise these ever-growing amounts of data to plan their operations and pricing, including information about the products individuals purchase, the links they visit on the Internet and the times they make phone calls. Decision-support queries are those that are utilised for these kinds of applications and they may require terabytes of data. Single-processor systems are not capable of handling such enormous amounts of data at the needed rates.
- Database queries are naturally set-oriented, which makes parallelisation easy. Parallel query processing has been shown to be powerful and scalable by several commercial and academic systems.
- Parallel machines have proliferated and become reasonably priced in tandem with the decreasing cost of microprocessors.

Motivation for Parallel Databases

With the growing use of computers, organisations' transaction needs have expanded. Furthermore, the expansion of the World Wide Web has led to the creation of several websites with millions of visitors and the growing volume of data gathered from these visitors has resulted in extraordinarily enormous databases at numerous businesses.

The needs of applications that must query extraordinarily large databases (of the order of petabytes, or 1000 terabytes, or equivalently, 1015 bytes) or process extraordinarily high volumes of transactions per second (of the order of thousands of transactions per second) are what motivate parallel database systems. Database systems that are centralised or client-server are insufficiently powerful to manage these kinds of applications.

In order to manage the enormous volume of users on the web, web-scale applications nowadays frequently need hundreds to thousands of nodes (and in some circumstances, tens of thousands of nodes). Businesses utilise these ever-growing amounts of data to plan their operations and pricing, including information about the products individuals purchase, the links they visit on the internet and the times they make phone calls. Decision-support queries are those that are utilised for these kinds of applications and they may require terabytes of data. Such massive amounts of data cannot be handled by single-node systems at the necessary rates.

Database queries are naturally set-oriented, which makes parallelisation easy. Parallel query processing has been shown to be powerful and scalable by several commercial and academic systems. Parallel computers are now widely used and reasonably priced due to the considerable decline in the cost of computing systems over time. Individual computers have evolved into multicore architectures-based parallel machines. Therefore, parallel databases are reasonably priced, even for tiny businesses. Although hundreds of node parallel database systems have been commercially available for several decades, the

Notes

number of such solutions has increased significantly during the mid-2000s. Widespread use has also been observed for open-source query processing and parallel data storage platforms like Hadoop Map-Reduce and Hive (among many others), as well as the Hadoop File System (HDFS) and HBase.

It is worth mentioning that application programs are typically constructed such that they can be executed in parallel on a number of application servers, which connect via a network with a database server, which may itself be a parallel system. In addition to data storage and query processing in databases, application programs can also be processed in parallel using the parallel architectures covered in this section.

Key Concepts of Parallel Databases:

Parallel Processing: Parallel databases exploit the concept of parallel processing, where operations are separated into subtasks that can be executed concurrently across numerous processors or nodes. The execution of queries and data processing are accelerated by this parallelism.

Parallel Query Execution: A parallel database divides the workload among several processors by executing queries concurrently. Big dataset retrieval and processing times are greatly shortened by this concurrent query execution.

Data Distribution: Data in parallel databases is split up among several nodes or partitions. Because each node is in charge of overseeing a portion of the data, queries can be executed in parallel thanks to parallel data processing.

Parallel Transaction Processing: Parallel databases provide parallel transaction processing in addition to queries. This improves the system's capacity to manage concurrent user activity by enabling the execution of numerous transactions at once.

Advantages of Parallel Databases

- Scale-up or increased throughput. High throughput can be attained via parallel databases, allowing them to manage several transactions or queries at once. They are hence a good fit for applications requiring high throughput.
- Enhanced reaction time (acceleration). The considerable increase in performance is the main benefit of parallel databases. Parallel databases process queries and transactions much quicker than typical single-node databases because they divide the burden among multiple processors.
- Beneficial for applications that need to query extraordinarily huge databases and handle a high volume of transactions quickly (thousands of transactions per second).
- Significant gains in performance.
- Enhanced system availability. Scalability is provided via parallel databases, which enable the system to accommodate additional nodes or processors. Organisations can extend their parallel database architecture to meet increasing needs as data volume and processing requirements increase.
- Increased adaptability Since the data is spread and processed concurrently in parallel databases, large datasets can be processed with efficiency. This guarantees best use of available resources and cuts down on the amount of time needed for intricate data processing.
- Able to accommodate a huge user base.

Disadvantages of Parallel Databases

- More start-up costs.

- Interference problem.
- Skew problem.

Notes

By processing queries and transactions in parallel, parallel databases provide unmatched performance and scalability—a significant breakthrough in database management. Parallel database adoption is expected to increase as long as organisations can manage the growing volumes of data and the need for real-time analytics. Parallel databases have a bright future ahead of them thanks to their continued integration with cloud computing, machine learning and hybrid architecture research and development. They are becoming an essential part of the data management and analytics environment.

I/O Parallelism

I/O parallelism, in its most basic form, is the process of dividing up relations across several discs in order to minimise the amount of time needed to retrieve them from disc. The most prevalent method of data partitioning in a parallel database environment is horizontal partitioning. The process of horizontal partitioning involves dividing, or declustered, a relation's tuples across numerous discs so that each tuple is stored on a single disc. Numerous partitioning techniques have been put forth.

Partitioning Techniques

We outline three fundamental data-partitioning techniques. Assume that the data needs to be partitioned among n discs, D_0, D_1,..., D_{n-1}.

Round-robin. This method sends the ith tuple to disc number Di $_{\text{mod n}}$ after scanning the relation in any order. Each disc has roughly the same number of tuples as the others thanks to the round-robin technique, which guarantees an even distribution of tuples across discs.

Hash partitioning. Using this declustering technique, the partitioning attributes are one or more characteristics from the schema of the supplied relation. A hash function with a range of {0, 1,..., n − 1} is selected. The partitioning properties are used to hash each tuple in the original relation. The tuple is stored on disc D_i if the hash function returns i.

Range partitioning. This method assigns contiguous attribute-value ranges to each disc in order to distribute tuples. It selects a partitioning vector [v_0, v_1,..., v_{n-2}] and partitioning attribute A such that, if i< j, then v_i <v_j. The division of the relation is as follows: Let t be a tuple where t[A] = x. T moves to disc D0 if x < v_0. T is placed on disc D_{n-1} if x >= v_{n-2}. T is placed on disc D_i+1 if $v_i \leq x < v_i$+1.

In range partitioning, for instance, tuples with values less than five might be assigned to disc 0, values between five and forty to disc 1 and values larger than forty to disc 2.

Comparison of Partitioning Techniques

We can retrieve a relation in parallel utilising all of the discs once it has been divided up among multiple discs. In a similar vein, a relation can be written in parallel to several discs during partitioning. As a result, with I/O parallelism, the transfer speeds for reading or writing a complete relation are substantially faster than without it. But there are other ways to get data than reading a relation in its entirety or scanning it. Data access can be categorised in the following ways:

1. Scanning the entire relation.
2. Finding a tuple associatively (e.g., employee name = "Campbell"); these searches, also known as point queries, look for tuples with a given value for a particular attribute.
3. Finding every tuple whose value of a particular property falls between a given range (say, 10000 < salary < 20000); these searches are known as range queries.

Notes

These kind of access are supported by various partitioning strategies with varying degrees of efficiency:

- Round-robin. Applications that want to read the whole relation sequentially for every query are best suited for this strategy. Since each of the n discs must be used for the search, this technique makes processing both range and point queries difficult.
- Hash partitioning. Point queries based on the partitioning attribute are the most appropriate use case for this method. For instance, if a relation is partitioned based on the telephone number attribute, we may use the partitioning hash algorithm to apply to 555-3333 and then search that disc to find the record of the employee with telephone number = 555-3333. By directing a query to one disc, you can avoid the startup costs associated with starting inquiries on several discs and free up space on the other discs to handle additional requests.

  Sequential searches over the full relation can also benefit from hash partitioning. There should be little variation in the number of tuples in each disc if the partitioning attributes constitute a key of the relation and the hash algorithm is an effective randomising function. Accordingly, the time needed to scan the relation in a single disc system is around 1/n times longer than the time needed to scan the relation.

  However, the method is not well suited for point queries on properties that are not partitioned. Range queries are also poorly served by hash-based partitioning since hash algorithms usually do not maintain proximity inside a range. In order to find the answers to range queries, all of the discs must be scanned.
- Range partitioning. Point and range queries on the partitioning attribute are a good fit for this method. To find the disc on which the tuple is located for point queries, we can refer to the partitioning vector. In range queries, the possible range of discs on which the tuples may exist is determined by consulting the partitioning vector. In each scenario, the search is limited to discs that include any potentially interesting tuples. One benefit of this feature is that the query is usually sent to one disc instead of all the discs if there are only a few tuples in the searched range. Range partitioning leads to better throughput while retaining good response time because other discs can be used to answer different queries. However, when there are a lot of tuples in the searched range (which occurs when the queried range makes up a bigger portion of the relation's domain), a lot of tuples must be fetched from a small number of discs, which causes an I/O bottleneck (hot spot) at those discs. This execution skew example has all processing happening in one or a small number of partitions. On the other hand, round-robin and hash partitioning would use all of the discs for these types of requests, resulting in a quicker response time for roughly the same throughput.

Moreover, joins and other relational operations are impacted by the type of partitioning. Therefore, the operations that must be carried out determine the partitioning strategy to choose. Round-robin partitioning is often not chosen over hash or range partitioning. The number of discs that should be used to divide a relation in a system with numerous drives can be selected as follows: It is preferable to assign a relation to a single disc if it has fewer tuples than what can fit into a single disc block. Large relations should ideally be divided among all of the discs that are accessible. A relation should be allotted min(m, n) discs if it consists of m disc blocks and the system has n discs available.

3.1.2 Design of Parallel Databases

When possible, a parallel database management system (DBMS) will operate on several processors or CPUs and is primarily intended to carry out query activities in parallel.

To get the same throughput as one huge machine would, a parallel database management system links several smaller machines together.

Notes

There are primarily three architectural concepts for parallel DBMSs in parallel databases. They are listed in the following order:

Shared Memory Architecture- Several CPUs are connected to a network of connections in a shared memory architecture. They can share common disc arrays and a single or global main memory. It should be mentioned that this design allows for the support of many CPUs with a single copy of a multithreaded operating system and multithreaded database management system. Furthermore, the robust coupled architecture of shared memory allows several CPUs to share memory. It is also known as Symmetric multiprocessing (SMP). This architecture supports a wide range of applications, starting with personal workstations that use RISC to support several microprocessors operating in parallel.

Advantages:

- For a restricted set of CPUs, it offers fast data access.
- There is effective communication.

Disadvantages:

- It is limited to using 80 or 100 CPUs concurrently.
- As the number of CPUs increases, the bus or the interconnection network becomes blocked.

Shared Disk Architectures:

Different CPUs are connected to a network of connections in a shared disc architecture configuration. Every CPU in this has access to the same disc and has its own memory. Additionally, take note that since there is no memory sharing among the CPUs, the operating system and database management system are duplicated on each node. Loosely connected architecture designed for centralised applications is known as shared disc architecture. Another name for them is clusters.

Advantages:

- With each CPU having its own memory, the connectivity network is no longer a bottleneck.
- Load balancing in shared disc architecture is simpler.
- Better fault tolerance is present.

Disadvantages:

- Interference and memory conflict issues become more prevalent as CPU counts rise.
- Additionally, there is a scalability issue.

Shared Nothing Architecture

A multiple processor design known as “shared nothing” allows each CPU to have its own memory and disc storage. In this, a node connects several CPUs to a network of connections. Additionally, keep in mind that no two CPUs can access the same disc space. There is no memory or disc resource sharing in this design. Massively parallel processing (MPP) is another name for it.

Advantages:

- Due to the lack of resource sharing, it is more scalable.

Notes

- It is possible to add more than one CPU.

Disadvantages:

- Because communications need data transmission and software interaction on both sides, their cost is higher.
- Compared to shared disc systems, non-local disc access is more expensive.

Hierarchical Architecture:

Combining shared disc, shared memory and shared nothing architectures results in this design. Thanks to the availability of several processors and more memory, this architecture is scalable. However, other architecture finds it expensive.

In order to improve performance, scalability and overall efficiency, designing parallel databases is a difficult undertaking that includes coordinating several processing units to work cooperatively on huge datasets. In order to split the burden across several processors or nodes and allow for the simultaneous execution of queries and transactions, parallel databases make use of parallel processing techniques. We will examine the fundamentals of designing parallel databases in this investigation, covering architecture, data distribution, parallel query processing, fault tolerance and performance-maximising concerns.

1. Architecture of Parallel Databases: A parallel architecture with many processing units, memory and storage components is the foundation upon which parallel databases are constructed. Shared-memory and shared-nothing are the two main architectural styles. Processors share a same memory area in a shared-memory architecture, facilitating direct communication and coordination. A shared-nothing design, on the other hand, minimises inter-node communication by using distributed nodes, each with its own memory and storage. The kind of workload, fault tolerance and scalability needs all influence the architecture decision.
2. Data Distribution Strategies: For parallel databases to fully utilise the processing capability of numerous nodes, efficient data distribution is essential. To distribute data uniformly among nodes, partitioning techniques based on hashing or ranges are frequently utilised. Using a hash function on a column, hash-based partitioning determines which node is in charge of storing each record. Data is partitioned using range-based partitioning according to predefined ranges of values in a column. In order to guarantee that every node has an equal burden, the objective is to reduce data skew.
3. Parallel Query Processing: The foundation of parallel databases is parallel query processing, which allows queries to be executed concurrently across numerous nodes. There are other ways to accomplish parallelism, such as inter- and intra-node parallelism. Using many processors within a node to perform a query concurrently is known as intra-node parallelism. By splitting up a query into smaller parts and distributing them among multiple nodes, inter-node parallelism enables each node to work independently on its own subset of data. The effectiveness of parallel query processing is increased by methods like parallel sort-merge join and parallel hash join.
4. Fault Tolerance Mechanisms: In order to guarantee continuous operation in the event of hardware malfunctions, node crashes, or other disturbances, fault tolerance is crucial in parallel databases. Redundancy is provided via data replication between nodes, which enables data availability even in the event of a node failure. In the event of a failure, automatic failover algorithms reroute queries to nodes that are healthy. Replaying transactions up to the point of failure through checkpointing and logging techniques guarantees that the system can recover from failures.

Notes

5. Query Optimisation and Execution Plans: A key component of designing a parallel database is query optimisation for concurrent execution. The query optimiser creates execution plans that account for the system's capacity for parallel processing. Efficient query execution is made possible by methods like parallel hash joins, parallel aggregation and parallel index searches. When creating execution plans, the optimiser has to take into account things like data distribution, inter-node communication costs and node capabilities.
6. Data Locality and Cache Coherency: In parallel databases, minimising data locality is crucial to cutting latency. In order to reduce the amount of inter-node communication required for query processing, data should be spread. By ensuring that every node uses its local data, cache coherency algorithms minimise the requirement for synchronisation and improve system performance overall. Depending on the needs of the application, consistency models like eventual consistency may be used.
7. Scalability and Dynamic Resource Allocation: Scalability is a crucial factor in the design of parallel databases since it enables the system to add more nodes in order to manage growing workloads. The optimum use of resources across nodes is ensured by dynamic resource allocation systems, which optimise their use. Load balancing techniques maximise the capacity for parallel processing and avoid performance bottlenecks by distributing queries evenly among nodes.
8. Monitoring and Performance Tuning: Sustaining optimal system health requires efficient systems for performance tuning and monitoring. Query response time, throughput and resource utilisation are examples of important performance metrics that administrators can set up to help them find possible bottlenecks and improve system performance. Automated tuning procedures make dynamic parameter adjustments in response to workload variations, guaranteeing that the system evolves to meet changing requirements.
9. Testing and Evaluation: Ensuring that a parallel database design is effective requires thorough testing and evaluation. Concurrency testing analyses the system's performance under concurrent transactions, whereas scalability testing measures the system's capacity to manage growing workloads. Furthermore, simulations and performance benchmarks aid in locating any problems and areas where the design has to be improved.
10. Future Considerations: Parallel databases will need to change to accommodate new hardware architectures and developing technologies. Distributed computation skills can be improved by integrating with parallel programming paradigms like MapReduce or by using parallel processing frameworks like Apache Spark. Future-proofing involves keeping up with advances in parallel computing, designing for flexibility to upcoming technologies and assuring compatibility with evolving hardware architectures.

Parallel database design is a complex process that includes architectural choices, scalability concerns, fault tolerance techniques, data distribution strategies and query optimisation. Large-scale data processing and analytical workloads are well suited for a well-designed parallel database system because it makes use of the cooperative processing capacity of several nodes to achieve excellent performance, scalability and fault tolerance.

3.1.3 Distributed Databases Principles

A distributed database system (DDBS) is a database that is physically kept on multiple computers located at various locations and linked by a communication network. Usually, a DBMS that can operate independently of the others is in charge of each location. To put it

Notes

another way, every site has its own local users, local DBMS and local data communications manager, making it a standalone database system site. Every site has its own local locking, logging and recovery software in addition to transaction management software. A distributed database system treats the entire database as a single collection of data, even though it is geographically scattered. All parts of the system, including query processing and optimisation, concurrency management and recovery, are significantly impacted by the placement of data items and the level of autonomy of individual sites.

A distributed database system divides data and transaction processing across one or more CPUs connected to a network, with each CPU serving a unique purpose within the system. The computers in the dispersed systems communicate with one another over various communication channels, such as high-speed networks of telephone lines. They don't share drives or primary memory. Applications can access data from both local and remote databases with the help of a distributed database system. Client/server architecture is used by distributed database systems to handle information requests. Workstations and mainframe systems are among the different types of computers that can be included in a distributed system. In a distributed database system, the computers are called by several various names, like sites or nodes. The general structure of a distributed database system is given in below Figure.

The requirement to provide local database autonomy at geographically dispersed locations gave rise to distributed database systems. For instance, local branches of large companies, national banks, or multinational banks may have separate branches with localised databases. The distributed database concept came about as a result of the development of communication and networking systems. It became feasible to provide communication across these dispersed systems, enabling efficient data access across computers situated in various geographic areas.

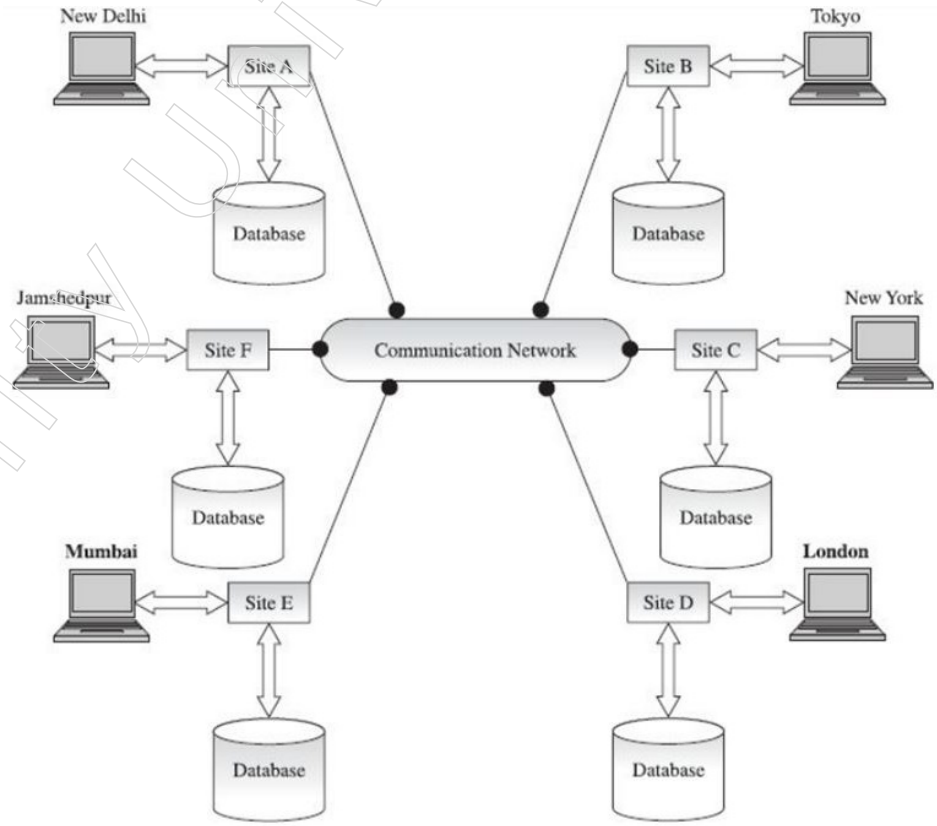


Figure: Distributed database architecture

Image Source: Database Management System, S K Sinha, Second Edition

Notes

Distributed database systems connect disparate geographically dispersed systems that already exist. Because of this, the machines at the various sites are probably heterogeneous, having completely distinct individual architectures. For instance, an ORACLE database system may be found at one site on a Sun Solaris UNIX system, a DB2 database system may be found on an OS/390 machine at another site and Microsoft SQL may be found on an NT machine at a third. A few examples of commercial distributed database management systems (DDBMS) are Oracle, DB2 and Ingres/Star.

Differences between Parallel and Distributed Databases

Geographically dispersed sites make up distributed databases. Compared to networks inside a single data centre, this results in lesser bandwidth, increased latency and a larger chance of failures for the network connections. Because distributed databases constitute the foundation of many systems, they require awareness of both the physical location of data and network problems and latency. Later in this section, we address these concerns. Specifically, it is frequently preferable to maintain a copy of the data at a data centre near the final user.

Node failure is a problem that is addressed by parallel database systems. However, some failures, especially those brought on by fires, earthquakes, or other natural disasters, have the potential to destroy a substantial number of nodes in a data centre as a whole. To provide high availability, distributed database systems must function even in the case of a data centre failure. To prevent all of the data centres from being impacted by a single natural disaster, data must be replicated among geographically dispersed data centres. While implementation specifics may vary, high availability strategies such as replication and others are comparable in parallel and distributed databases.

Distributed databases can be independently managed, allowing for a certain amount of operational autonomy at each location. These databases are frequently the outcome of merging pre-existing databases to enable cross-database transactions and inquiries. On the other hand, central administration may be possible for dispersed databases designed to provide geographic distribution as opposed to databases created by combining pre-existing databases. While nodes in parallel databases often have similar capacities, nodes in distributed databases typically vary greatly in size and function.

There are two types of transactions in a distributed database system: local and global. A local transaction only retrieves information from the nodes from which it originated. In contrast, a global transaction is one that accesses data from multiple nodes or from a node other than the one where it was started.

Table: Differences between Parallel and Distributed Databases

Feature	Parallel Database	Distributed Database
Definition	uses multiple processors to process a single, large task concurrently	Involves multiple databases geographically distributed
Organisation	Focus on breaking down a single task into smaller, manageable pieces	Focus on supporting multiple independent darabase
Data Distribution	Centralised data with all processors accessing a shared dataset	Data is spread across multiple locations or nodes
Communication	Tightly controlled communication optimised for specific tasks	Involves network communicationbetween distributed nodes
Fault Tolerance	Often limited to the failure of individual processors	More comprehensive, addressing network, node, and site failures

Notes

Use Cases	Suited for tasks that can be parallelised on a centralised dataset	Suit4ed for scenarios where data needs to be distributted geographically
Examples	Parallel processing of a large analytical query	Multi-site enterprise applications with distributed data
Coordination	Focus on efficient coordination for parallel task execution	Emphasises coordination for data consistency across locations
Scalability	Typically scales within a shared dataset	Scales by adding more nodes or databases across locations

Desired Properties of Distributed Databases

Transparency in the impact of data distribution is a goal of distributed database systems. The following characteristics should be included in distributed database systems:

- Distributed data independence.
- Distributed transaction atomicity

Distributed Data Independence

Distributed data independence attribute enables users to ask inquiries without identifying where the reference relations or copies or fragments of the relations, are located. Logical and physical data independence naturally lead to this principle. Moreover, cost-based optimisation should be applied methodically to queries that span many sites, accounting for variations in local computing costs and communication expenses.

A key idea in the planning and administration of distributed databases is distributed data independence. It describes the capacity to change how data is delivered and arranged in a distributed database system without having an impact on the queries or application programs that use the data. Achieving distributed data independence is essential for performance optimisation, allowing system maintenance without interfering with application operation and responding to changing requirements. We will examine distributed data independence in detail in this comprehensive examination, covering its types, obstacles and methods of attainment.

Types of Distributed Data Independence:

Logical Data Independence: Protecting application programs and queries from modifications to the distributed database's logical structure or schema is known as logical data independence. This implies that changes to the characteristics that are added or removed, table relationships, or data organisation should not affect the outward view that apps see.

Physical Data Independence: Application programs and queries are protected from modifications to the physical distribution or storage structures of data by physical data independence. This include adjustments to indexing strategies, data dissemination across various distributed environment nodes, or data storage format updates. Such adjustments should not impact application programs.

Achieving Logical Data Independence:

Abstraction through Views: Using views is one method for attaining logical data independence. Views offer a virtualised version of the data that may be customised to meet the needs of particular applications. By designing and utilising views, changes to the underlying schema can be performed without changing the outward view given to applications.

Notes

Data Definition Language (DDL): Changes to the database schema can be made without affecting the applications if the data definition language is strong enough. This entails modifying table architecture, adding or removing tables, or shifting relationships while maintaining consistency with the outward view.

Data Independence Layers: Logical data independence can be achieved by implementing layers of abstraction between the database schema and the application programs. In order to translate external requests into database operations without disclosing the underlying schema changes, middleware or data access layers might serve as mediators.

Achieving Physical Data Independence:

Data Distribution Strategies: Achieving physical data independence requires selecting suitable data distribution techniques, including replication or splitting. This makes it possible to alter how data is distributed among nodes without affecting how apps use the data.

Storage Management: Application programs should be able to see how storage management features like data compression and indexing are implemented. The external view of the data does not need to alter in order for these mechanisms to be adjusted or optimised.

Query Optimisation: Achieving physical data independence requires optimising distributed query processing algorithms and query execution schedules. Data distribution changes should not negatively impact query execution; instead, they should be clear.

Challenges in Distributed Data Independence:

Data Consistency: One of the biggest challenges is making sure that changes to the schema or distribution of the data do not affect the consistency of the data. Consistency must be preserved by using techniques like distributed transactions and concurrency control methods.

Query Optimisation: It is difficult to optimise queries in a distributed setting while being unaffected by modifications to data distribution or storage architectures. Cost-based optimisation and sophisticated query optimisation strategies become essential.

Schema Evolution: In order to manage schema evolution—that is, add or remove entities, attributes, or relationships—without breaking applications, versioning or compatibility methods must be put in place and careful planning must be done.

Strategies for Achieving Distributed Data Independence:

Metadata Management: The system can adjust to changes with ease when metadata is managed well, including dependencies and schema information cataloguing. The task of controlling and tracking schema change is aided by metadata repositories.

Versioning and Compatibility: Schema versioning can be implemented gradually by keeping compatibility layers intact and implementing versioning techniques. This guarantees that, until they are upgraded, current apps can function using the old schema.

Schema Evolution Tools: To achieve dispersed data independence, technologies that automate schema evolution procedures and give administrators an intuitive interface to handle schema modifications should be developed. Both logical and physical changes should be handled by these instruments.

Future Considerations:

Adaptive Systems: More distributed data independence may be attained when

Notes

distributed database systems develop towards more flexible and self-governing designs. Adaptive systems are able to adapt their schema, query patterns and data distribution dynamically.

Machine Learning and Automation: Enhancing the system's capacity to attain and sustain data independence can be accomplished by utilising automation and machine learning techniques to anticipate and handle changes in the dispersed environment. The process of adaptation can be streamlined by automation.

Global Data Management: As distributed databases reach beyond geographically scattered locations, global data management solutions must be implemented to assure data consistency and independence. This involves taking regulatory compliance, security and data privacy into account.

A challenging but crucial component of developing and running distributed database systems is achieving distributed data independence. Applications are protected from changes in the database schema, data distribution, or storage structures by logical and physical data independence. Overcoming obstacles, putting techniques into practice and anticipating trends all help to create resilient distributed database systems that can easily change to meet changing needs while preserving application data independence.

Distributed Transaction Atomicity

Users can create transactions that access and update data at several sites in the same way that they would write transactions over only local data thanks to the distributed transaction atomicity attribute. Specifically, a transaction's effects between sites ought to stay atomic. In other words, if the transaction commits, all modifications are preserved and if it aborts, none are preserved.

For distributed database systems to be reliable and consistent, distributed transaction atomicity is essential. One of the ACID qualities (Atomicity, Consistency, Isolation, Durability) is atomicity, which ensures that a transaction is handled as a single, indivisible work unit. Ensuring atomicity presents special difficulties in a distributed context, when data is dispersed among several nodes. We will examine distributed transaction atomicity in detail in this examination, discussing its significance, difficulties and methods used to get it.

Definition of Atomicity in Distributed Transactions:

Atomicity Principle: A transaction is either carried out in its whole or not at all thanks to atomicity. To maintain data consistency, the transaction is rolled back to its starting point if any portion of it fails. This idea can be applied to actions involving several nodes in a distributed environment, guaranteeing that all nodes either commit or abort the transaction together.

Importance of Atomicity in Distributed Transactions:

Data Consistency: In distributed databases, atomicity plays a critical role in preserving data consistency. It makes sure that the database maintains consistency even in the face of faults or failures, avoiding incomplete updates that can cause inconsistent data.

Reliability and Integrity: Because distributed systems sometimes span several locations, it is crucial to guarantee the integrity and dependability of transactions. Atomicity ensures that a transaction will either fail without leaving any trace or succeed, leaving the system in a valid state.

Isolation from Concurrent Transactions: The isolation provided by the atomicity attribute keeps concurrent transactions from interfering with one another. Atomicity guarantees

Notes

that the result of one transaction does not influence the outcome of other transactions until it is committed in a distributed system where several transactions may be processing concurrently across nodes.

Recovery from Failures: Software bugs, node malfunctions and network problems are all common in distributed systems. By guaranteeing that the system can roll back partial transactions in the case of a failure and preserve a consistent state, atomicity streamlines the recovery procedure.

Challenges in Achieving Distributed Transaction Atomicity:

Network Failures and Communication Delays: Network outages and latency can affect node-to-node communication in a distributed system. Ensuring that all nodes either commit or abort a transaction involves techniques to handle communication problems and delays.

Partial Failures: Atomicity is challenged by partial failures, in which some nodes commit while others abort because to mistakes or crashes. It becomes difficult to coordinate a consistent result across all nodes, particularly in heterogeneous systems.

Concurrency Control: Robust concurrency control mechanisms are necessary for the concurrent execution of transactions across distant nodes. Achieving atomicity requires making sure that transactions are isolated from one another and that disputes are handled properly.

Data Replication and Consistency Models: It might be difficult to keep consistency between copies during a transaction in distributed databases that use data replication. The coordination necessary for atomicity is affected by consistency models, such as eventual consistency or strong consistency.

Strategies for Achieving Distributed Transaction Atomicity:

Two-Phase Commit (2PC): A popular mechanism for attaining atomicity in distributed transactions is called Two-Phase Commit. To decide whether to commit or cancel the transaction, all participating nodes must coordinate with a coordinator node. In spite of its simplicity, 2PC is prone to the blocking coordinator problem and can experience blocking problems.

Three-Phase Commit (3PC): An expansion of 2PC that tackles some of its shortcomings is called Three-Phase Commit. In order to manage uncertainty and enhance fault tolerance, it adds a second phase. However, in some circumstances, 3PC might still encounter blockage.

Optimistic Concurrency Control: Transactions can continue with no locks at all thanks to optimistic concurrency control. It is assumed that disagreements are uncommon and that they are found and settled during the commit stage. This strategy can boost concurrency but may lead to additional rollbacks in case of conflicts.

Quorum-Based Systems: In distributed databases with data replication, quorum-based systems employ the idea of quorums to decide a majority vote for committing or aborting a transaction. This technique enables flexibility and fault tolerance, especially in settings with network partitions.

Distributed Transaction Managers: Centralising the coordination of dispersed transactions is facilitated by the use of distributed transaction managers. These managers take care of the intricate details of atomicity, making sure that every involved node comes to an agreement and completes the transaction as a single atomic unit.

Notes

Future Considerations and Emerging Technologies:

Blockchain and Distributed Ledgers: Proof-of-work and proof-of-stake are two examples of decentralised consensus processes that are introduced by distributed ledger and blockchain technologies. These protocols offer a basis for attaining atomicity in a decentralised and trustless fashion.

Edge Computing and IoT: Distributed transactions face new hurdles as edge computing and Internet of Things (IoT) devices proliferate. Innovative approaches specifically designed for these situations are needed to maximise atomicity in edge contexts with resource limitations.

Machine Learning for CoordinationIt seems promising to use machine learning methods to dynamically coordinate remote transactions. The effectiveness of reaching atomicity may be improved by adaptive systems that are able to pick up on and adjust to the peculiarities of the dispersed environment.

For distributed database systems to preserve data consistency, dependability and integrity, atomicity in transactions is a challenging but essential goal. To overcome the obstacles presented by partial failures, concurrency and network failures, strong coordination protocols like Two-Phase Commit or Three-Phase Commit must be implemented. The future of distributed transaction atomicity may be shaped by factors like blockchain, edge computing and machine learning as technologies advance, offering creative answers to the particular problems presented by distributed environments.

3.1.4 Architectures of Parallel Databases

Multiple central processing units (CPUs) are connected to a computer system in a parallel database architecture. For parallel devices, there are various architectural models available. Those that are most well-known are mentioned below:

- Shared-memory multiple CPU.
- Shared-disk multiple CPU.
- Shared-nothing multiple CPU.
- Shared- Hierarchical multiple CPU

Shared-memory Multiple CPU Parallel Database Architecture

A computer with numerous (multiple) concurrently operating CPUs connected to an interconnecting network can share (or access) a single (or global) main memory and a shared array of disc storage in a shared-memory system. A single copy of a multithreaded operating system and multithreaded database management system can therefore handle several CPUs in a shared-memory architecture. A shared-memory multiple CPU architecture's schematic diagram is displayed in the figure below. Although far quicker in performance than a single-CPU of the same power, the shared-memory architecture of parallel database systems is most similar to the conventional single-CPU processor of centralised database systems. When it comes to attaining moderate parallelism, this structure is appealing. It has been comparatively simple to transfer numerous commercial database systems to shared-memory architectures.

The processors in a shared-memory design can access a common memory, usually via an interconnection network. The processors share discs as well. The advantage of shared memory is that it facilitates incredibly effective communication across processes. Any process can access data in shared memory without the need for software relocation. utilising memory writes, which typically take less than a microsecond, allows a process to

communicate with other processes significantly more quickly than utilising a communication mechanism.

Notes

These days, mobile phones as well as desktop PCs frequently have 4–8 core multicore processors. As of 2018, the number of cores in high-end processing systems, like Intel's Xeon processor, has increased consistently to approximately 72. On the other hand, the Xeon Phi coprocessor systems have about 72 cores, with up to 8 CPUs on a board. Because integrated circuit characteristics, such logic gates, have been getting smaller over time, more gates may be placed on a single chip, which explains why there are more cores in each chip. About every one and a half to two years, the number of transistors that can fit on a given surface of silicon doubles.

It makes reasonable to put several processors on a single chip because the number of gates needed for a CPU core has not increased accordingly. An on-chip processor is referred to as a "core" in order to retain the distinction between on-chip multiprocessors and conventional processors. As a result, a machine is said to have a multicore processor.

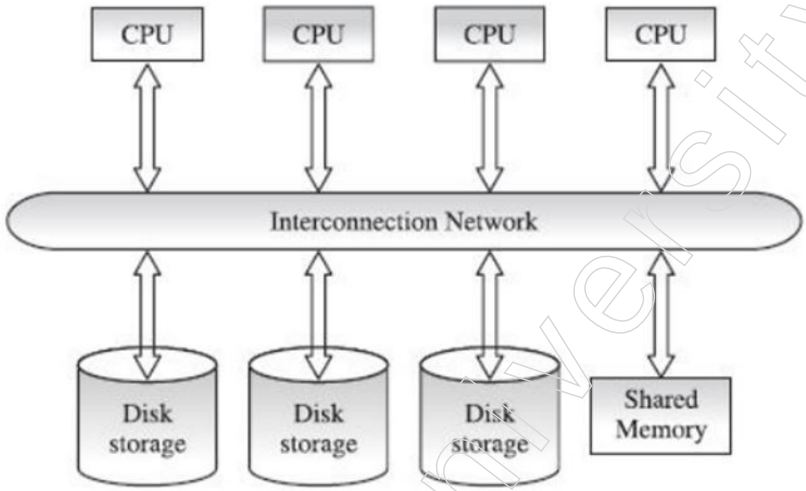


Figure: Shared-memory multiple CPU architecture

Image Source: Database Management System, S K Sinha, Second Edition

A single processor's cores usually access the same memory. Additionally, a system may contain several CPUs that are able to share memory. A further consequence of the growing gate count has been a consistent growth in main memory capacity and a reduction in main memory cost per byte. In recent years, shared-memory parallel processing has gained importance due to the low-cost availability of multicore processors and the concurrent availability of vast amounts of memory at low cost.

Benefits of Shared-memory Architecture

- CPUs communicate with each other quite effectively. Any CPU can access data without the need for software relocation. utilising memory writes, which typically take less than a microsecond, allows a CPU to communicate with other CPUs much more quickly than utilising a communication mechanism.
- Because operating system services can be employed to make advantage of the extra CPUs and main memory can be used for this purpose, there are minimal communication overheads.

Limitations of Shared-memory Architecture

- A extremely fast technique is used for memory access and it is challenging to divide without sacrificing performance. Therefore, extra care must be taken in the design

Notes

to ensure that each CPU has equal access to the shared memory. Furthermore, a parallel CPU should not unexpectedly alter the data that it has retrieved.

- Shared-memory architecture is not scalable to more than 80 or 100 CPUs in parallel since each CPU shares the communication bus or interconnection network. As the number of CPUs rises, the bus or the interconnection network becomes a bottleneck.
- When new CPUs are added, the CPUs have to wait for their turn to access memory on the bus.

It is a difficult task to harness the computational power of several processors to collectively execute queries and transactions concurrently in a shared-memory, multiple CPU parallel database system. Multiple CPUs with separate execution capabilities and a shared memory space that enables smooth communication and data exchange between these processors are the fundamental components of this architecture. By doing away with the requirement for explicit data transfer procedures, this shared-memory solution facilitates effective inter-process communication and improves the database system's overall performance. The architecture involves several key components, including a Memory Management Unit (MMU) responsible for translating virtual addresses into physical addresses within the shared memory, a Database Buffer Pool that enables caching of data pages to reduce redundant reads and a Lock Manager crucial for coordinating synchronisation of access to shared data structures.

The distribution and division of data over several CPUs is one of the architecture's key considerations. Efficient data distribution algorithms, such as hash or range partitioning, maintain a balanced workload, preventing hotspots and minimising communication overhead. Data partitioning algorithms need to be properly created in order to accomplish load balancing and enhance overall database performance. Moreover, the architecture's key components include parallel processing and query execution. Subtasks are created from queries and each CPU handles its allocated subtask in parallel and independently. In order to avoid conflict and provide smooth parallel execution, a query coordinator distributes subtasks and guarantees effective coordination. By utilising parallelism to increase efficiency, techniques like parallel hash joins and aggregations optimise complex query procedures.

Maintaining consistency and cache coherency are essential components of shared-memory architectures. To guarantee that changes made by one CPU are visible to others, robust cache coherency mechanisms are needed. Various strategies, including cache invalidation or snooping, may be used. The selection of consistency models, which specify the visibility of updates across CPUs, must take performance considerations and application requirements into account. In order to minimise performance overhead and guarantee data consistency, the design must find a balance.

Another important factor to take into account is scalability; the architecture should be built with the flexibility to add more CPUs in order to handle growing workloads. By allowing the system to adjust to changing workloads, dynamic resource allocation algorithms maximise resource utilisation and reduce bottlenecks. Data availability and system recoverability are guaranteed in the event of hardware failures or unforeseen circumstances by fault tolerance and reliability measures, such as data replication and checkpointing.

To effectively parallelise work within individual CPUs, integration with parallel programming paradigms, such as OpenMP or POSIX threads (Pthreads), is necessary. By offering structures for expressing parallelism in the code, these programming models let developers make the most of the shared-memory architecture. Furthermore, the creation of parallel database APIs abstracts the underlying intricacies, facilitating smooth developer interaction with the parallel database system.

Notes

An advanced method for improving database performance and scalability is a shared-memory, multiple CPU parallel database architecture. To enable effective parallel execution of queries and transactions, it makes use of shared memory regions, many CPUs working together and thoughtfully constructed components. This architecture is well-suited for applications needing high-performance database systems that can handle complicated workloads and big datasets since it takes into account important factors including data distribution, query optimisation, cache coherency, scalability and fault tolerance.

Shared-disk Multiple CPU Parallel Database Architecture

A shared disc system consists of several CPUs connected to a network of interconnections, each with its own memory, but with access to a shared array of discs or, more frequently, the same disc storage. The capacity and throughput of the interconnection network mechanism have a major role in determining the system's scalability. Every node has a copy of the DBMS and the operating system since memory is not shared by CPUs. Since all nodes have access to the same data, it's feasible that two or more nodes will wish to read or write the same data simultaneously. Therefore, in order to guarantee the preservation of data integrity, some sort of global (or distributed) locking method is needed. A parallel database system is another term occasionally used to describe the shared-disk architecture. A schematic diagram of a shared-disk multiple CPU architecture is shown in the picture below.

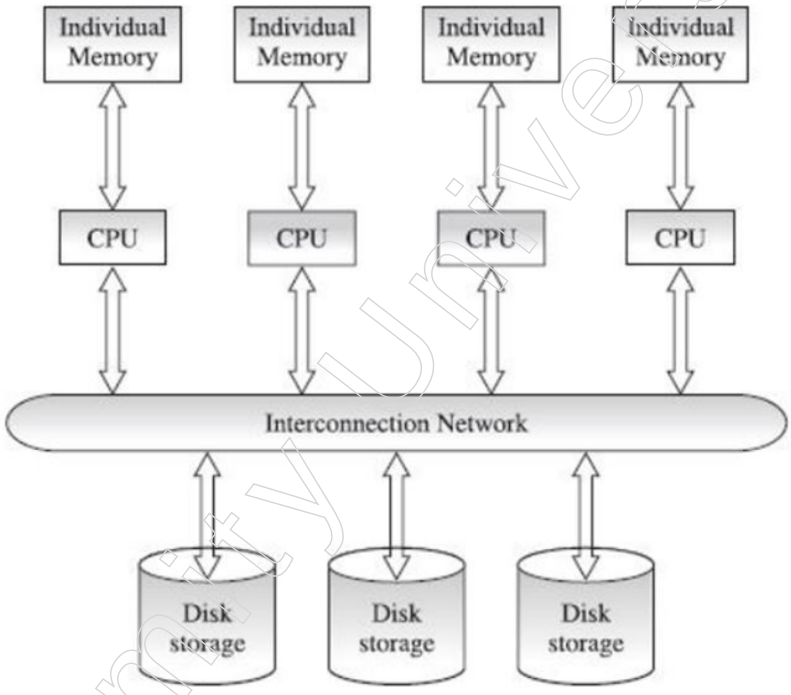


Figure: Shared-disk multiple CPU architecture

Image Source: Database Management System, S K Sinha, Second Edition

Benefits of Shared-disk Architecture

- Load balancing is made easier with shared-disk architecture because data is not constantly distributed across the available CPUs.
- There is no bottleneck on the memory bus because every CPU has its own memory.
- It provides a reasonably priced way to give some fault tolerance. Since the database is stored on discs that are accessible from all CPUs, in the event of a CPU or memory failure, the other CPUs take over the operation.

Notes

- It is now widely accepted in many applications.

Limitations of Shared-disk Architecture

- As the number of CPUs rises, shared-disk architecture likewise has similar issues with interference and memory contention bottleneck. Existing CPUs become slower as more are added due to greater competition for memory access and network bandwidth.
- The scalability of shared-disk architecture is another issue. A bottleneck occurs in the connection to the disc subsystem, especially when the database uses the discs frequently.

When designing a database system with several CPUs and a shared disc, a complex structure must be created so that the processors may work together to access a centralised storage disc. Each CPU in this architectural concept has a distinct memory region, but they all have common access to a central disc that houses the database. This shared-disk solution offers a mix between performance, scalability and flexibility, making it especially useful for managing complicated workloads, massive datasets and scenarios with high concurrency. We will examine all of the important parts, factors and difficulties related to a shared-disk multiple CPU database design in this in-depth analysis.

Multiple CPUs with the capacity to carry out tasks separately are at the core of this design. These CPUs work together cooperatively, sharing the workload among themselves while they process queries and transactions simultaneously. A key element is the shared disc, which functions as a centralised storage location for database files. This makes data continuously available to all CPUs, doing away with the need for redundant copies and streamlining system-wide data exchange. Coordination and information exchange are made easier by an effective interconnect or communication fabric, which guarantees smooth connection between CPUs.

In order to convert virtual addresses used by the CPUs into physical locations on the shared disc, the Memory Management Unit (MMU) is essential. To guarantee appropriate memory access and allocation in the shared-disk context, this translation is necessary. Furthermore, every CPU has its own memory-based Database Buffer Pool that it uses to cache frequently visited data pages in order to reduce disc input/output (I/O) and improve query efficiency. An essential component of the architecture is a lock manager, which controls concurrency to maintain data consistency and organises access to shared data structures.

When using a shared-disk multiple CPU architecture, effective data distribution techniques are essential. To achieve consistent data distribution and avoid hotspots, methods like range partitioning and hash partitioning are used. A key component is the execution of queries in parallel, where each CPU processes a query independently into smaller jobs. In order to orchestrate the parallel execution of queries, a Query Coordinator distributes subtasks in an effective manner and avoids conflict. By utilising parallelism to increase overall efficiency, techniques such as parallel hash joins and aggregations optimise complex query procedures.

Data consistency between CPUs is mostly dependent on cache coherency techniques. Snooping and cache invalidation are two strong strategies used to guarantee that changes made by one CPU are visible to other CPUs. Application needs and performance concerns determine which consistency models—strict consistency, sequential consistency, or eventual consistency—to use.

Notes

Scalability is a crucial factor and the architecture is made to support growing workloads by supporting the addition of additional CPUs. By allowing the system to adjust to changing workloads, dynamic resource allocation algorithms maximise resource usage. In the event of a failure, fault tolerance methods like as data mirroring, replication, checkpointing and recovery guarantee data availability and system recoverability.

Communication and coordination between CPUs in a shared-disk environment are facilitated by integration with parallel programming paradigms, such as the Message Passing Interface (MPI). With the help of parallel database APIs, developers may easily communicate with shared-disk databases by using a high-level interface that abstracts the underlying complexities.

To ensure the best possible system health, monitoring and performance adjustment are essential. Efficient operation is ensured by defining key performance measures and putting in place automated performance tuning methods that dynamically modify settings based on changes in workload. Thorough documentation and developer and administrator training programs improve knowledge and proficiency in system management.

Finding possible bottlenecks and guaranteeing effective management of fluctuating workloads depend heavily on testing and evaluation, including concurrency and scalability testing. Future-proofing strategies ensure adaptation to changing requirements and technological improvements. Examples of these strategies include planning for flexibility and compatibility with emerging technologies.

For a database system, a shared-disk, multiple CPU design offers a reliable way to manage complicated workloads and big datasets. This architecture offers the required flexibility, scalability and high-performance parallel processing capabilities by effectively utilising the cooperative computing capability of numerous CPUs sharing a central storage disc. To successfully design and implement a shared-disk multiple CPU database architecture, careful consideration of components, data distribution schemes, cache coherency mechanisms, scalability factors and fault tolerance measures is essential.

Every node in the shared-disk architecture has its own CPU and memory, but through an interconnection network, every node can directly access every disc. Compared to a shared-memory design, this architecture has two benefits. Compared to a shared-memory system, a shared-disk system can support more CPUs. It also provides an affordable means of supplying a certain level of fault tolerance. Since the database is stored on discs that are accessible from every node, if one node fails, the other nodes can take over its responsibilities.

By employing a RAID architecture, we may make the disc subsystem itself fault resistant, enabling the system to continue operating even in the event that one or more discs fail. I/O parallelism is also partially enabled by a RAID system's huge number of storage devices.

Large banks of storage devices, or discs, are connected to nodes that require the data via a storage-area network (SAN), a high-speed local area network (see below Figure). The storage devices offer a view of a logical disc, or collection of discs, that conceals the specifics of the underlying discs, even though they are physically made up of an array of several discs. A logical disc, for instance, might be far larger than any of the physical discs and its size can be expanded by adding more physical discs. Even if the discs are physically apart, the processing nodes can access them as if they were local discs.

Notes

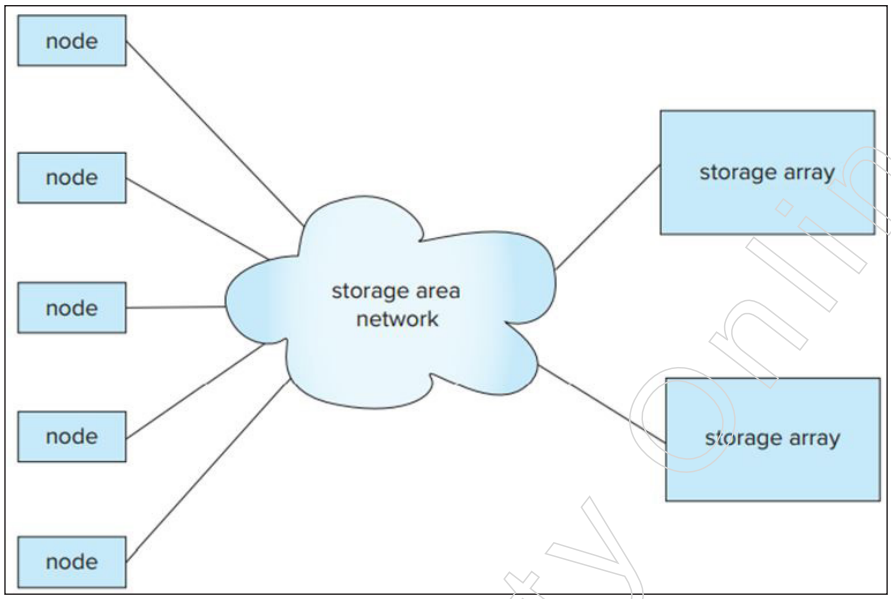


Figure: Storage-area network

Image Source: Component Database Systems, Klaus R. Dittrich andreas Geppert

In order to ensure that the network continues to operate even in the event of a component failure, such as a link or connection to the network, storage-area networks are typically constructed with redundancy, such as numerous pathways between nodes. Shared-disk systems are ideally suited for storage-area networks. Storage-area networks combined with shared disc architecture have become popular in applications that don't require a lot of parallelism but do need high availability. Shared-disk systems can support more processors than shared-memory systems since they don't require specialised hardware for communication, but communication between nodes takes longer (up to a few milliseconds) because it must travel across a communication network. The bandwidth of the network connection used to access storage in a shared-disk system is typically smaller than the bandwidth available to access local storage, which is one of the systems' limitations. Scalability may be hampered as a result of storage access acting as a bottleneck.

Shared-nothing Multiple CPU Parallel Database Architecture

Every CPU in a shared-nothing system has local memory and disc storage, but no two CPUs can access the same disc storage region. Instead, several CPUs are connected to a network of connections via a node. A fast connectivity network facilitates all CPU-to-CPU communication. On its own disc or discs, the node serves as the server for its own data. Hence, there is no memory or disc resource sharing in shared-nothing situations. Every CPU is equipped with a separate copy of the operating system, DBMS and a subset of the data under DBMS management. An architecture with shared-nothing multiple CPUs is schematically depicted in Figure. CPUs that share control over database services in this kind of architecture typically distribute the data among themselves. Then, by splitting up the task and corresponding via message over the high-speed network (at a rate of megabits per second), CPUs carry out transactions and queries.

In order to create a shared-nothing, multiple CPU parallel database architecture, one must create a complex framework in which several processors work together to separately perform transactions and queries, each with its own set of resources and storage. Large-scale data processing jobs and intricate analytical queries are best suited for this architecture. We will examine all of the important elements, factors and difficulties related to a shared-nothing multiple CPU database design in this in-depth investigation.

Notes

Multiple CPUs, each with its own memory, storage and processing capabilities, form the basis of this architecture. Shared-nothing architectures, in contrast to shared-memory architectures, spread data over several nodes, with a dedicated storage system on each node. This distribution encourages parallelism and scalability by reducing contention and enabling each processor to work independently on its own subset of data. The design is made simpler since shared memory eliminates the requirement for intricate cache coherency techniques.

In the shared-nothing design, every node functions independently, handling a subset of the data and running queries individually. By introducing additional nodes to the system, this decentralised method enables horizontal scalability by doing away with the requirement for a centralised coordinator. Fault tolerance is improved by the absence of common resources between nodes, which means that the failure of one node does not affect the system as a whole.

Effective data distribution techniques are essential in an architecture with shared-nothing many CPUs. Partitioning strategies that are range-based or hash-based are frequently used to divide data equally among nodes, avoiding hotspots and guaranteeing a balanced workload. It is essential to execute queries in parallel, with each CPU handling a specific subset of data in a separate and concurrent manner. In order to prevent performance bottlenecks and guarantee that each node receives a comparable workload, load balancing procedures are essential.

Because the data in a shared-nothing architecture is distributed, query optimisation becomes a challenging problem. When creating execution plans, optimisers have to take into account things like data distribution, network latency and local processing capacity. Pushdown predicates and distributed joins are two strategies used to reduce data transfer between nodes and improve query performance.

In a shared-nothing architecture, cache coherency is made easier because every node works independently on its own local data. Depending on the needs of the application, consistency models like eventual consistency may be used. Because there aren't any shared resources, there is less conflict and more system efficiency.

One key feature of shared-nothing architectures is scalability. The system's smooth scalability to accommodate growing workloads is made possible by the addition of additional nodes. The optimum use of resources across nodes is ensured by dynamic resource allocation systems, which optimise their use.

A fundamental feature of shared-nothing architectures is fault tolerance. Because the system is distributed, the database as a whole is not jeopardised when one node fails. Redundancy is ensured via data replication between nodes and system reliability is improved by features like automatic failover and recovery plans.

Distributed computation across nodes is facilitated by integration with parallel processing frameworks like Apache Spark or programming models like MapReduce. With the help of parallel database APIs, developers may easily communicate with shared-nothing databases by using a high-level interface that abstracts the underlying complexities.

Maintaining optimal system health requires tools for performance adjustment and monitoring. Query response time, throughput and resource utilisation are examples of important performance metrics that administrators can set up to help them find possible bottlenecks and improve system performance. Parameters are dynamically adjusted by automated tuning methods in response to variations in workload.

Notes

To make sure the system can manage growing workloads and concurrent transactions effectively, testing and evaluation—including concurrency and scalability testing—are essential. The system's lifespan and usefulness are ensured by future-proofing considerations, such as designing for compatibility with growing hardware architectures and adaptability to emerging technologies.

The multiple CPU parallel database design is a potent remedy for workloads involving extensive data processing and analysis. This architecture uses the collaborative processing capability of several CPUs across dispersed nodes, giving scalability, fault tolerance and parallelism. An effective shared-nothing multiple CPU database architecture requires careful consideration of its component parts, data distribution plans, query optimisation methods and fault tolerance controls.

Within a shared-nothing system, a CPU, memory and one or more discs are present in every node. A high-speed network of connections allows the nodes to communicate. For the data on the disc or discs that it possesses, a node serves as the server. The shared-nothing approach circumvents the drawback of requiring all I/O to pass through a single interconnection network because local disc references are handled by local discs at each node. Furthermore, shared-nothing interconnection networks—like the tree-like interconnection network—are typically made to be scalable, meaning that when more nodes are added, their transmission capacity grows. As a result, shared-nothing systems are easier to scale and can accommodate a very high number of nodes.

Since transmitting data requires software activity at both ends, the fundamental disadvantages of shared-nothing systems are greater communication and nonlocal disc access costs than in a shared-memory or shared-disk design. Shared-nothing architectures are popular for handling extremely large data volumes because of their great scalability, which can support scalability to thousands of nodes or, in extreme situations, even tens of thousands of nodes.

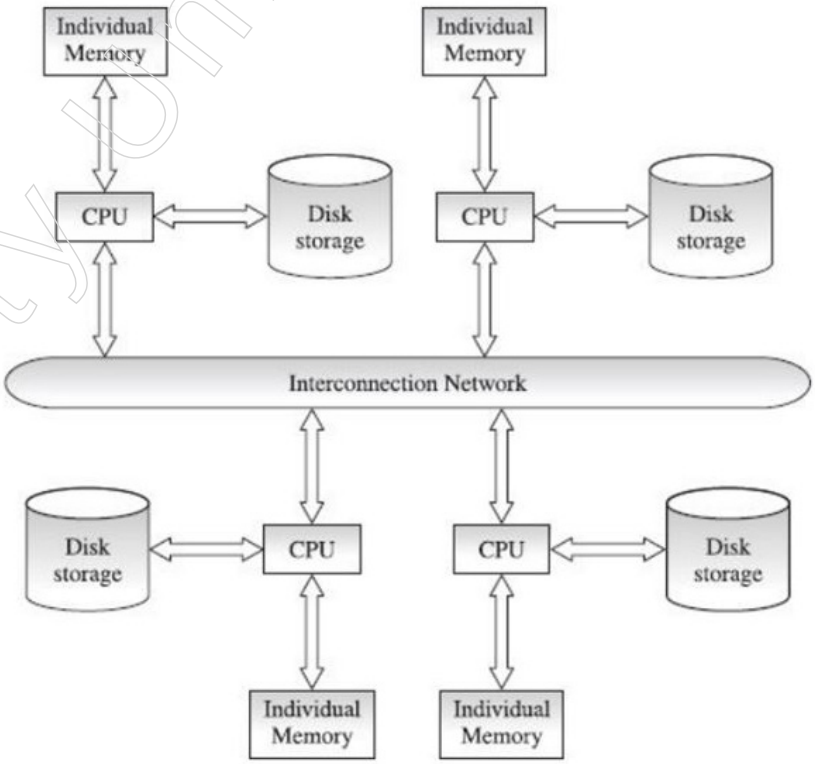


Figure: Shared-nothing multiple CPU architecture

Image Source: Database Management System, S K Sinha, Second Edition

Notes

Benefits of Shared-nothing Architecture

- Because shared-nothing systems do not share resources, they reduce CPU contention and provide great scalability.
- Since local disc references are serviced by local discs at each CPU, the shared-nothing design avoids the restrictions of forcing all I/O to go through a single interconnection network. The only things that go across the network are queries, disc accesses that are not local and result relations.
- In shared-nothing designs, the interconnection networks are typically made to be scalable. The system can therefore develop (or scale) in a way that is proportionate to the power and capacity of the newly added components by adding more CPUs and discs. This allows for practically linear scalability, giving users a significant return on their investment in new hardware (resources). Stated differently, linear speed-up and linear scale-up are provided by shared-nothing architecture.
- As more nodes are added, the shared-nothing architecture's transmission capacity increases due to its linear speed-up and scale-up qualities, making it easily able to serve a large number of CPUs.

Limitations of Shared-nothing Architecture

- Load balancing shared-nothing systems is challenging. To ensure that all system resources are utilised effectively, the system workload must be divided in many multi-CPU environments. An administrator must correctly partition or divide the data across the several discs in order to split or balance this workload across a shared-nothing system and ensure that each CPU is kept roughly as busy as the others. It is challenging to accomplish this in real life.
- When new CPUs and discs are added to a shared-nothing architecture, the data may need to be redistributed to take use of the additional hardware (resource), necessitating a more thorough restructuring of the DBMS code.
- The costs of communication and non-local disc access are higher than in shared-disk or shared- memory architecture since transmitting data includes software activity at both ends.
- Due to speed-of-light constraints, the size of high-speed networks is constrained. This means that having CPUs that are physically close to one another is necessary for a parallel design. Local area network (LAN) is another name for this network design.
- A single point of failure is introduced into the system via shared-nothing designs. Since each CPU is in charge of its own disc or discs, if one of these CPUs fails, the data on one or more of these discs becomes unaccessible.
- It needs an operating system that can handle the large volume of messaging needed to facilitate communication across processors.

Applications of Shared-nothing Architecture

- Relatively inexpensive CPU technology is highly suited for shared-nothing architectures. Because of the system's excellent scalability, customers can begin with a comparatively small and inexpensive system and add more relatively inexpensive CPUs as needed to satisfy capacity requirements.
- Mega parallel processing systems are based on the shared-nothing paradigm.

Shared- Hierarchical Multiple CPU

The shared-memory, shared-disk and shared-nothing architectures' features are combined in the hierarchical architecture. The system's nodes, which do not share memory

Notes

or discs, are linked together via an interconnection network at the top level. Thus, a shared-nothing architecture represents the top level. In reality, every system node may be a few CPUs running a shared memory system. Another option is for every node to function as a shared-disk system and for every system that shares a set of discs to function as a shared-memory system.

Therefore, a system might be constructed as a hierarchy, with a shared-nothing architecture at the top and a shared-memory architecture with a few processors at the base, possibly with a shared-disk design in the centre. A hierarchical architecture with shared-memory nodes linked in a shared-nothing architecture is shown in the figure below. Parallel database systems today often function on a hierarchical architecture, where each node supports shared-memory parallelism, with numerous nodes interconnected in a shared-nothing fashion.

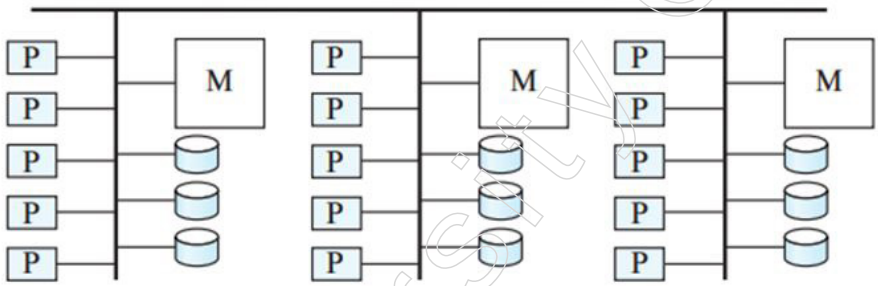


Figure: Shared- Hierarchical multiple CPU

Image Source: Component Database Systems, Klaus R. Dittrich andreas Geppert

In order to process queries and transactions in a hierarchical structure, several processors work together to provide a sophisticated framework in shared-hierarchical, multiple CPU parallel database architecture design. The goal of this architecture is to achieve a compromise between shared-memory architectures' centralised control and decentralised shared-nothing systems. We will examine all of the essential elements, factors and difficulties related to a shared-hierarchical multiple CPU database design in this in-depth analysis.

Multiple CPUs, each with its own memory, storage and computing power, form the basis of this architecture. Unlike the entirely decentralised approach of shared-nothing architectures, a shared-hierarchical model includes a hierarchical structure where nodes are organised into tiers or levels. While nodes at upper levels may coordinate and aggregate findings from nodes in lower tiers, each node within a tier works independently on its own data. An approach to distributed processing that is more regulated and structured is made possible by this hierarchical structure.

Effective data distribution techniques are essential in an architecture with shared-hierarchical multiple CPUs. It is possible for nodes in the same tier to share some data, which promotes cooperation and parallel processing there. It is imperative to use caution while managing the distribution of data among tiers to prevent bottlenecks and guarantee an even burden. To maximise data distribution, methods like range-based or hash-based partitioning may be used while taking the hierarchical structure into account.

A basic feature is parallel query execution, where each CPU independently processes its allotted portion of data in parallel. Each tier needs load balancing mechanisms in order to guarantee that the burden is split equally among the nodes. Moreover, higher levels include coordination methods to control the flow of results and requests throughout the hierarchy.

Because of the tiered structure, query optimisation in a shared-hierarchical design becomes a challenging problem. In addition to local processing power and data distribution

Notes

inside a tier, optimisers also need to account for inter-tier communication cost. Techniques like cost-based and distributed query optimisation are used to create effective execution plans that take the hierarchical structure into account.

Cache coherency inside tiers is simpler than across tiers, as each node within a tier functions independently on its own data. However, depending on the needs of the application, consistency models must be carefully selected. The introduction of higher-level coordinating mechanisms could guarantee global uniformity among layers.

One feature that sets shared-hierarchical designs apart is scalability. As workloads increase, the hierarchical structure facilitates smooth scalability both inside and across levels. At every level of the hierarchy, effective use of the resources that are available is ensured via dynamic resource allocation algorithms.

Because nodes in one tier may not be impacted by tier-level failures, the hierarchical structure is inherently fault-tolerant. Fault tolerance is improved via redundancy methods, such as data replication between and between tiers. The overall dependability of the system is enhanced by recovery techniques at every tier.

Distributed computation across tiers is facilitated by integration with distributed computing frameworks like Apache Spark and parallel programming models like MapReduce. Parallel Database APIs isolate the underlying intricacies, offering developers with a high-level interface to interact with the shared-hierarchical database system effortlessly.

Maintaining optimal system health requires tools for performance adjustment and monitoring. Query response time, throughput and resource utilisation are examples of important performance metrics that administrators can set up to help them find possible bottlenecks and improve system performance. Automated tuning mechanisms take the hierarchical structure into account and dynamically modify settings in response to variations in workload.

To make sure the system can manage growing workloads and concurrent transactions effectively, testing and evaluation—including concurrency and scalability testing—are essential. The system's lifespan and usefulness are ensured by future-proofing considerations, such as designing for compatibility with growing hardware architectures and adaptability to emerging technologies.

A multiple CPU parallel database design that is shared-hierarchical is an example of a sophisticated strategy that blends the advantages of decentralisation and structured hierarchical organisation. This design provides scalability, fault tolerance and parallelism in a regulated and organised way by utilising the cooperative processing capability of numerous CPUs both inside and between tiers. A shared-hierarchical multiple CPU database architecture must be carefully designed and implemented, taking into account data distribution strategies, query optimisation methods and fault tolerance controls.

3.1.5 Design: Implementation of Parallel Databases

Using parallel processing architectures and techniques to improve database operations' performance is known as parallel database implementation. Here are important actions and things to think about when setting up parallel databases:

Data Partitioning: To distribute data among several nodes in a parallel system, partition the database. Various partitioning techniques, including range-based, hash-based and list-based techniques, can be utilised according to the type of data and query patterns.

Notes

Divide queries into smaller jobs that can be completed concurrently to achieve parallel query execution. This comprises aggregation, JOIN and select operations parallelisation. To achieve effective parallelisation, parallel algorithms and query strategies must be used.

Index building and maintenance can be done simultaneously by utilising parallel indexing techniques. This is critical for query performance optimisation, particularly in situations where indexes are critical to query execution.

Enable data ingestion and loading in parallel to guarantee effective data intake into the database. This is especially crucial in situations involving data warehousing, when rapid loading of massive volumes of data is required.

Strategies for Distributing Data:

Select a suitable data distribution plan in accordance with the parallel system's architecture. Data replication, data partitioning and combinations of these tactics are common techniques.

Using fault-tolerant techniques will help you deal with failures in a parallel setting. In order to make sure the system is robust in the face of hardware or software failures, this entails techniques like data replication, checkpointing and recovery procedures.

Processing Transactions in Parallel:

Expand the use of parallelism in transaction processing to enable the effective execution of concurrent transactions. Implement isolation levels and transaction control techniques that are consistent with parallel architectures.

Make use of query optimisation strategies created especially for parallel databases. This entails selecting suitable parallelisation techniques for certain query patterns, optimising join algorithms and taking into account the expense of data transfer between nodes. Specialised parallel data warehouse designs that are intended for parallel processing should be taken into consideration. Massively Parallel Processing (MPP) systems, which can grow horizontally by adding more nodes to the system, are frequently included in these architectures.

Resource Management: To distribute and reallocate resources among parallel nodes, use efficient resource management techniques. In order to avoid resource competition and bottlenecks, this involves optimising memory consumption, CPU allocation and I/O activities. Monitoring of Scalability and Performance: Set up systems for monitoring the parallel database system's scalability and performance. Analyse system performance on a regular basis, look for bottlenecks and scale the system to handle increasing workloads.

Database management systems that are expressly built to take advantage of parallel processing capabilities are called parallel database management systems, or DBMSs. Teradata, Amazon Redshift and Apache Greenplum are a few examples.

3.1.6 Fragmentation

During the DDBS design phase, the most popular methods for dividing a database into logical units and storing specific data across several sites are data fragmentation and data replication. A fundamental idea in distributed databases, fragmentation deals with how data is dispersed among several nodes in a networked setting. Databases in distributed systems are frequently dispersed over several geographical regions or nodes in order to increase fault tolerance, improve scalability and improve access times. The partitioning and distribution of data is determined by fragmentation algorithms, which impact the system's ease of use, performance and efficiency. We will examine every facet of fragmentation

in distributed databases in this thorough investigation, covering types of fragmentation, implementation tactics, difficulties and effects on system properties.

Notes

Strategies for Implementation

Hash-Based Fragmentation: Applying a hash function to a column's data and figuring out where each fragment should be kept is known as hash-based fragmentation. By distributing data evenly among nodes, this technique reduces hotspots and ensures effective load balancing. When there is uncertainty in the patterns of data access, hash-based fragmentation works well.

Range-Based Fragmentation: With range-based fragmentation, data is divided into groups according to predetermined ranges of values within a column. Data may be divided, for instance, according to a given time frame or numerical range. When there are inherent divides in the data or when queries frequently focus on particular ranges of values, range-based fragmentation can be advantageous.

Round Robin Fragmentation: Round robin fragmentation is a process that requires cyclically dispersing data among nodes in an equitable manner. To ensure equitable distribution, each node gets the subsequent piece of data in a circular sequence. Round robin fragmentation is easy to apply and works well in situations where fragmentation criteria are not specified.

Directory-Based Fragmentation: With directory-based fragmentation, data fragments are mapped to their appropriate locations via a centralised directory. You use this directory to find the position of a specific fragment. Even while it adds a certain amount of centralisation, it offers flexibility and manageability, particularly in situations where data distribution is dynamic.

Challenges in Fragmentation:

Query Performance: The efficiency of fragmentation techniques has a direct bearing on query performance. Inadequate fragmentation design can result in more data being transferred between nodes, which can raise latency and cause queries to execute more slowly than they should.

Data Skew: When some fragments receive a disproportionate amount of data in comparison to others, this is known as data skew. Performance bottlenecks may result from this imbalance since skewed data nodes may have heavier loads and longer reaction times.

Dynamic Workloads: It might be difficult to adjust to workloads that are dynamic and involve changing data access patterns over time. Strategies for fragmentation must be adaptable enough to take into account changes in the needs for data distribution without necessitating constant modification.

Consistency and Synchronisation: It is critical to provide synchronisation and consistency amongst distributed fragments. Coordination issues may arise when changes made to one fragment must be communicated to others, particularly when several nodes are able to update the same piece of data.

Impact on System Characteristics:

Scalability: System scalability is improved by effective fragmentation, which makes it possible to add nodes to handle increasing demands. Specifically, horizontal fragmentation can make parallel processing easier and increase overall scalability.

Fault Tolerance: Partitioning that is well-designed can help with fault tolerance. Data availability and system resilience are ensured by the ability to access replicas or portions of data stored on other nodes in the event of a node failure.

Notes

Data Locality: Data locality—the degree to which data resides near the nodes that require access to it—is influenced by fragmentation tactics. By enhancing data localisation, location-based fragmentation lowers network latency and boosts system performance overall.

Ease of Maintenance: The maintenance-friendliness is impacted by the fragmentation approach selected. Strategies that entail intricate coordination mechanisms or call for regular adjustments could make administering and maintaining the distributed database more difficult.

Future Considerations and Emerging Trends:

Machine Learning for Dynamic Fragmentation: There is potential in using machine learning methods to regulate fragmentation dynamically based on changing workloads. Optimising fragmentation tactics over time using adaptive systems could lead to improved performance.

Blockchain and Smart Contracts: Decentralised consensus techniques for managing fragmentation are introduced by integrating blockchain technology and smart contracts into distributed databases. This could offer more security and openness when distributing data.

Edge Computing and Fragmentation: The emergence of edge computing presents fresh difficulties as well as chances for fragmentation. To maximise data access and administration, fragmentation solutions must be designed to work with the decentralised nature of edge environments.

A distributed system's fault tolerance, scalability and performance are significantly influenced by the complex idea of fragmentation in distributed databases. Crucial factors to take into account include selecting suitable fragmentation techniques, dealing with issues like data skew and query performance and adjusting to changing workloads. The future landscape of fragmentation in distributed databases could be shaped by the integration of machine learning, blockchain and edge computing as technology advances, offering creative solutions to the problems associated with distributed data management.

Data fragmentation is the process of dividing a database into logical units that can be allocated for storage at different locations. A relation may be partitioned (or fragmented) into multiple fragments (pieces) for physical storage purposes in data fragmentation and each fragment may have multiple replicas. These pieces of information are complete enough to rebuild the original relationship. Every piece that makes up a certain relation will stand alone. Not a single fragment can be deduced from the others, nor does it possess any limitations or projections that allow for derivation from other fragments. For example, let us examine an EMPLOYEE relation as stated in below table.

Table: Relation EMPLOYEE

Relation: EMPLOYEE			
EMP-ID	*EMP-NAME*	*DEPT-ID*	*EMP-SALARY*
E-106519	Kumar Abhishek	4	55000
E-112233	Thomas Mathew	5	45000
E-123456	Alka Parasar	2	60000
E-100600	Jose Martin	5	30000
E-198900	Meena Singh	4	80000
E-224569	Avinash Parasar	2	70000

Notes

This relationship can now be divided into the following three pieces:

FRAGMENT EMPLOYEE AS

MUMBAI_EMP AT SITE 'Mumbai' WHERE DEPT-ID = 2

JAMSHEDPUR_EMP AT SITE 'Jamshedpur' WHERE DEPT-ID = 4

LONDON_EMP AT SITE 'London' WHERE DEPT-ID = 5;

The fragmented relation mentioned above will be stored at multiple locations, as illustrated in the figure below. The records or rows associated with the employees from "Mumbai" (DEPT-ID = 2) are kept at the Mumbai site, while the employees from "Jamshedpur" (DEPT-ID = 4) are kept at the Jamshedpur site and the employees from "London" (DEPT-ID = 5) are kept at the London site. The internal fragment names of the distributed database system are MUMBAI_EMP, JAMSHEDPUR_EMP and LONDON_EMP, as can be seen in this example. Appropriate JOIN and UNION operations are used to reconstruct the original relation.

Fragmentation independence, often known as fragmentation transparency, is something that a system that facilitates data fragmentation ought to facilitate. Therefore, it makes no sense for users to be concerned about fragmentation. It should appear to the users as though the data are not fragmented at all.

The user is shielded from awareness of the data fragmentation by DDBS. Stated differently, fragmentation independence indicates that users will see a representation of the data where the pieces are rationally reassembled using appropriate JOINs and UNIONs. The system optimiser is in charge of figuring out which fragments must be physically accessed in order to fulfil each individual user request. The two distinct methods for breaking up a relation are as follows:

- Horizontal fragmentation
- Vertical fragmentation
- Mixed fragmentation

Horizontal Fragmentation

A subset of the tuples (rows) in a relation that have every attribute is called a horizontal fragment of that relation. By giving each tuple or group (subset) of tuples in a relation a logical meaning, horizontal fragmentation divides the relation "horizontally" into one or more fragments. Afterwards, distinct locations within the distributed system can be assigned to these pieces. Defining a predicate that applies a restriction to the tuples in the relation results in a horizontal fragmentation. The relational algebra's SELECT function is used to define it.

One way to characterise a horizontal fragmentation is:

$\sigma P (R)$

where σ = relational algebra operators for selection

p = predicate based on one or more attributes of the relation

R = a relation (table)

The fragmentation example of the following graphic is horizontal and it can be expressed as follows in relational algebra:

Notes

MUMBAI_EMP :σDEPT-ID=2 (EMPLOYEE)

JAMSHEDPUR_EMP :$\sigma^{DEPT\text{-}ID=4}$(EMPLOYEE)

LONDON_EMP :σDEPT-ID=5 (EMPLOYEE)

The relational procedures of limitation are corresponding to horizontal fragmentation. The UNION operation is used in horizontal fragmentation to recreate the original relationship.

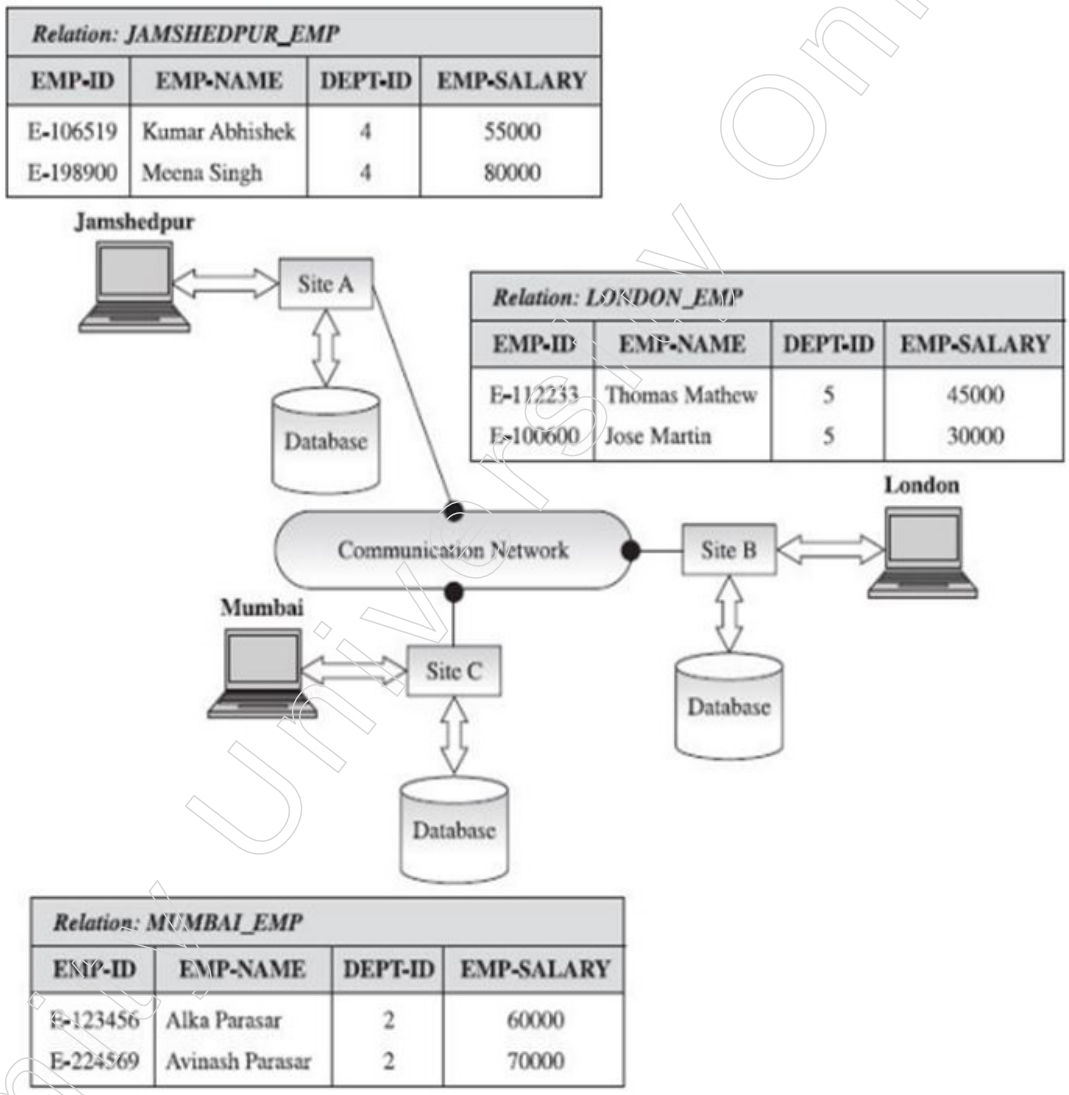


Figure: An example of horizontal data fragmentation

Image Source: Database Management System, S K Sinha, Second Edition

Vertical Fragmentation

By breaking the relationship down "vertically" into columns (attributes), vertical fragmentation divides the association. Because not every site requires every feature of a relation, a vertical fragment of a relation preserves only a subset of the relation's attributes at a given site. As a result, vertical fragmentation relates the features that are utilised jointly by significant transactions. When the two fragments (attributes) are stored independently, a straightforward vertical fragmentation is not entirely appropriate. We are unable to reconstruct the original employee tuples since the two fragments lack a common attribute.

Notes

In order to reconstruct the entire relation from the fragments, the main key (or candidate key) characteristics must be present in each vertical segment. The stored relation serves as the address for connecting tuples in vertical fragmentation, while the system-provided "tuple-ID" (or TID) serves as the primary key (or candidate key) attribute. Vertical fragmentation is described as follows and relates to the relational activities of projection:

∏ a1,a2…,an(R)

Where ∏ = relational algebra operator for projection

a1…,an = attributes of the relation

R = a relation (table)

As an illustration, the relation EMPLOYEE in the preceding table can be divided vertically as follows:

```
FRAGMENT EMPLOYEE AS
 MUMBAI_EMP (TID, EMP-ID, EMP-NAME) AT SITE 'Mumbai'
 JAMSHEDPUR_EMP (TID, DEP-ID) AT SITE 'Jamshedpur'
 LONDON_EMP (TID, EMP-SALARY) ATSITE 'London';
```

The following describes how the aforementioned vertical fragmentation will be stored at different locations, as illustrated in the figure below: the attributes (TID, EMP-ID, EMP-NAME) for employees in "Mumbai" are stored at the Mumbai site; the attributes (TID, DEPT-ID) for employees in "Jamshedpur" are stored at the Jamshedpur site; and the attributes (TID, EMP-SALARY) for employees in "London" are stored at the London site. The original relation is reconstructed using the JOIN technique.

MUMBAI_EMP : $\prod_{TID,\ EMP\text{-}ID,\ EMP\text{-}NAME}$ (EMPLOYEE)

JAMSHEDPUR_EMP : $\prod_{TID,\ DEP\text{-}ID}$ (EMPLOYEE)

LONDON_EMP : $\prod_{TID,\ EMP\text{-}SALARY}$ (EMPLOYEE)

Figure: An example of vertical fragmentation

Image Source: Database Management System, S K Sinha, Second Edition

Mixed Fragmentation

Occasionally, database schema fragmentation—either vertically or horizontally—is not enough to sufficiently disseminate the data for certain applications. Mixed or hybrid fragmentation is needed instead. Thus, horizontal (or vertical) fragmentation of a relation, followed by subsequent vertical (or horizontal) fragmentation of some of the fragments, is called mixed fragmentation. The relational algebra's projection (PROJECT) and selection (SELECT) operations are used to design a mixed fragmentation. Combining JOIN and UNION operations yields the original relation. Given is a mixed fragmentation.

$$\sigma_p(\prod_{a_1,\ a_2,\ldots.a_n}(R))$$

or

$$\prod_{a_1,\ a_2,\ldots.a_n}(\sigma_p(R))$$

The relation EMPLOYEE was vertically split in the vertical fragmentation example as

Notes

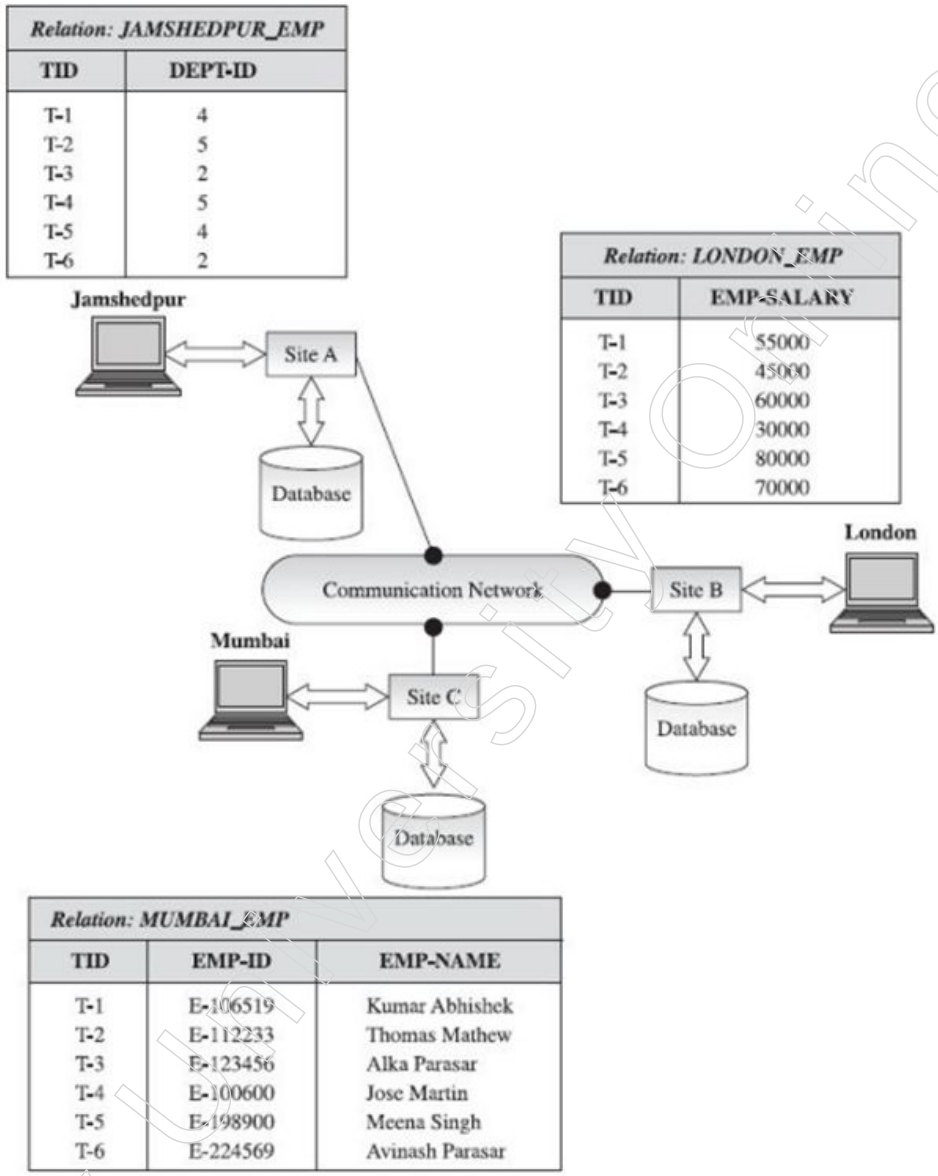


Figure: An example of vertical fragmentation

image Source: Database Management System, S K Sinha, Second Edition

S1 = MUMBAI_EMP : $\prod_{\text{TID, EMP-ID, EMP-NAME}}$(EMPLOYEE)

S2 = JAMSHEDPUR_EMP : $\prod_{\text{TID,DEP\_ID}}$(EMPLOYEE)

S1 = LONDON_EMP : $\prod_{\text{TID, EMP- SALARY}}$ (EMPLOYEE)

According to DEPID, we might now horizontally partition S2 in the manner described below:

$$S_{21} = \sigma_{DEPT\text{-}ID=2}(S_2) \quad : \quad \sigma_{DEPT\text{-}ID=2}(\text{JAMSHEDPUR\_EMP})$$

$$S_{22} = \sigma_{DEPT\text{-}ID=4}(S_2) \quad : \quad \sigma_{DEPT\text{-}ID=4}(\text{JAMSHEDPUR\_EMP})$$

$$S_{23} = \sigma_{DEPT\text{-}ID=5}(S_2) \quad : \quad \sigma_{DEPT\text{-}ID=5}(\text{JAMSHEDPUR\_EMP})$$

3.1.7 Transparencies in Distributed Databases

In essence, a distributed database is one that spans various sites, such as numerous computers or a network of computers, rather than being restricted to at least one server.

Notes

Distributed database systems exist on different locations that don't share any physical parts. When a certain database needs to be accessible by numerous individuals worldwide, this can be necessary. It needs to be handled with the users in mind. It resembles a single database.

In database management systems, transparency refers to the division of high-level system semantics from low-level implementation concerns. High-level semantics stands for the endpoint user and low level implementation concerns with intricate hardware implementation of data or how the data has been stored in the database. Transparency in DBMSs can be achieved by the use of data independence in different database layers.

Distributed databases have a feature called distribution transparency, which keeps users from knowing the internal workings of the distribution. Tables can be broken up into smaller pieces, duplicated and stored at various locations by the DDBMS designer. Users find the distributed database to be just as user-friendly as any centralised database, nevertheless, because they are unaware of these intricacies.

The dissemination of information to the user from the system in a transparent manner is referred to as transparency in DDBMS. It aids in concealing the information that the user must apply. Assume, for illustration, that data independence in a typical database management system (DBMS) is a type of transparency that aids in concealing from the user changes to the definition and structure of the data. However, their overall goals are the same. This implies that the distributed database should be used in the same way as a centralised database.

Transparencies in distributed databases refer to the degree to which the underlying complexity of a distributed system are hidden from users and applications. Transparency seeks to offer a smooth and uniform experience, protecting users and applications from the complexities of communication protocols, data distribution and system faults. We will examine many forms of transparency in distributed databases in this investigation, along with their importance, implementation methodologies and obstacles to overcome.

DDBMSs deal with communication networks, replication and data fragments, in contrast to traditional DBMSs. Therefore, these three elements are likewise involved in transparency.

- Transaction transparency
- Performance transparency
- DBMS transparency
- Distribution transparency

Transaction Transparency-

This openness ensures that distributed database regularity and integrity are maintained during all distributed transactions. Also, it essential to note that distribution transaction access is the data held at different sites. It's also important to note that every sub-transaction must be maintained atomicity by the DDBMS. (By this, we mean that the complete transaction must occur directly or not at all). Because DBMS fragmentation, allocation and replication structures are used, it is extremely complex.

Performance Transparency-

Because of this transparency, a DDBMS must function differently from a centralised database management system. Furthermore, since its architecture is dispersed, there shouldn't be any performance hiccups with the system. Similar to this, a distributed query processor—which is capable of translating a request for data into an ordered series

Notes

of actions on the local database—must be included in a DDBMS. The fragmentation, replication and allocation structure of DBMSs add additional layer of complexity to this.

Concurrency Transparency:

Concurrency transparency ensures that consumers and applications are insulated from the intricacies of concurrent access to data by numerous transactions. Concurrent transactions are frequently handled by distributed databases and methods including locking schemes, optimistic concurrency control and isolation levels are necessary to achieve transparency in concurrency management. It should be possible for users to engage with the system without having to actively handle any conflicts that might occur from running many tasks at once.

Failure Transparency:

Failure transparency is keeping users and applications unaware of the effects of node malfunctions or communication mistakes. Nodes in a distributed system can malfunction as a result of software bugs, hardware malfunctions, or network troubles. Failure transparency can be attained by recovery methods, automatic failover and fault-tolerant replication. Even in the event of node failures, users ought to have continuous access to data and services.

Migration Transparency:

In a distributed system, migration transparency represents the flow of information or services between nodes. Nodes may join or depart the network in dynamic contexts, requiring the movement of services or data. Mechanisms for updating distributed metadata, preserving consistency throughout migration and guaranteeing that users and programs can access data without being aware of its travel between nodes are all necessary to achieve migration transparency.

Heterogeneity Transparency:

In a distributed database system, heterogeneity transparency refers to concealing the variations in hardware, software, or communication protocols across nodes. Achieving transparency in heterogeneous settings guarantees that applications and users can interact with the system without being influenced by the heterogeneity of the underlying infrastructure. Transparency in heterogeneity is achieved by the use of protocol adapters, data translation layers and middleware.

Security and Privacy Transparency:

Transparency in security and privacy refers to keeping users and apps unaware of the specifics of authentication, authorisation and encryption systems. Ensuring secure data access and safeguarding user privacy are critical in distributed databases. Encryption methods, secure communication routes and access control mechanisms that function without explicit user intervention are necessary to provide transparency in security and privacy.

DBMS transparency-

Because it conceals the possibility that the local DBMS may differ, this transparency is only applicable to heterogeneous forms of DDBMS (Databases with several sites and varying operating systems, products and data models). Among all the transparencies, this one is the hardest to use as a generalisation.

Distribution transparency-

If a DDBMS exhibits distribution data transparency, the user does not need to be aware

that the data is fragmented because it aids in the user's recognition of the database as a single entity or logical system.

Notes

There are five different types of distribution transparency, which are covered below:

Fragmentation Transparency

In this form of transparency, the user doesn't have to know about fragmented data and, owing to which, it leads to the reason why database accesses are based on the global schema. This is comparable to SQL view users in that the user may not be aware that they are using a view of a table rather than the table itself. The transparency of fragmentation allows users to query any table as though it were unfragmented. As a result, it conceals the reality that the table the user is executing a query on is, in reality, a union of several fragments. It also hides the fact that the pieces are scattered across several locations.This is comparable to SQL view users in that the latter could not be aware that they are utilising a view rather than the actual table.

Transparency in fragmentation divides data among several nodes in an abstract way. Data in distributed databases is frequently broken up into smaller pieces for better scalability and parallel processing. Mechanisms that let users and apps engage with the system without being aware of how data is divided up or delivered are necessary to achieve fragmentation transparency. To achieve this transparency, directory-based techniques, adaptive fragmentation mechanisms and hybrid fragmentation strategies are used.

Location Transparency

If the DDBMS offers this kind of transparency, then the user must be aware of the ways in which the data has been fragmented; but, knowledge of the data's location is not required. The ability to query on any table or table fragment as if it were locally stored on the user's site is ensured by location transparency. The end user should not be aware that the table or any of its pieces are stored in a distributed database system at a remote location. Both the remote site(s) address and the access methods are concealed completely. DDBMS should have access to an accurate and up-to-date data dictionary as well as a DDBMS directory that includes information on the locations of data in order to provide location transparency.

In a distributed database system, location transparency seeks to abstract the actual locations of resources and data. Applications and users engage with the system without realising which node is handling their requests or where data is kept. In order to guarantee that users can access data without disclosing its precise location, location transparency is achieved through the use of methods like load balancing, distributed naming services and data replication.

Replication Transparency

The user is unaware that fragments are being copied when there is replication transparency. Failure and concurrency transparency are linked to replication transparency. Every time a user updates a data item, all copies of the table reflect the change. But the user shouldn't be aware of this process. Transparency in replication guarantees that users are not aware of database replication. It lets users query a table as though there were just one copy of the table.

Concurrency and failure transparency are related to replication transparency. Every time a user modifies a data item, all copies of the table automatically reflect the modification. But the user shouldn't be aware of this process. Transparency in concurrency is this. Furthermore, in the event that a website fails, users can continue utilising duplicate

Notes

copies to complete their requests without being aware that anything has gone wrong. Transparency has failed in this case.

Replication transparency is keeping the possibility of data being in numerous copies across many nodes hidden. Replication is frequently used to balance load, provide fault tolerance and enhance data availability. Replication transparency guarantees data access for users and programs even when they are unaware of the number of copies or their storage locations. Replication transparency is facilitated by synchronisation protocols, conflict resolution techniques and consistency methods.

Local Mapping Transparency

The user must specify the fragment names and the locations of the data items in local mapping transparency, taking into consideration any potential duplications. In DDBMS transparency, this is a more challenging and time-consuming query for the user.

Naming Transparency

We already know that there are two types of centralised database systems: DDBMS and DBMS. This implies that every entry in the database needs to have a distinct name. It also implies that the DDBMS needs to ensure that no database object with the same name is created by two sites. There are two approaches to address the issue of naming transparency: one is to establish a central name server whereby objects within the system can be given unique names; alternatively, we can add an object that begins with the creator site's identity.

Challenges in Achieving Transparencies

- Performance Overhead: There may be performance costs associated with implementing transparency methods, particularly when it comes to collaboration and communication. It can be difficult to strike a balance between performance and transparency.
- Consistency and Synchronisation: It is difficult to keep distributed copies consistent when they are fragmented or replicated. Mechanisms for synchronisation should protect data integrity without sacrificing openness.
- Scalability: It gets harder to achieve transparency as distributed databases grow in size. One important consideration is designing transparent procedures that scale well with the growing size of the distributed system.
- Dynamic Environments: Complexity arises when adjusting to dynamic situations where nodes can join or depart the network. Ensuring system consistency and updating metadata demands strong procedures to achieve transparency in the face of dynamic changes.

3.1.8 Transaction Control in Distributed Database

Transaction control is concerned with assigning and regulating the sites required for executing a transaction in a distributed database system. When it comes to deciding where to process the transaction and how to assign the centre of control, there are numerous choices available, such as:

- It is possible to designate one server as the control centre.
- It is possible for the centre of control to move between servers.
- There is a possibility that multiple servers will share control.

Notes

Coordinating the execution of transactions across several nodes in a networked environment is a complex yet crucial component of distributed database management. The main objective is to uphold the core ACID concepts of Atomicity, Consistency, Isolation and Durability—even in the distributed setting, where a number of issues need to be resolved, including concurrency, atomicity amongst nodes, consistency and failure handling. This in-depth investigation will cover the techniques, methods and intricacies related to transaction control in distributed databases.

A program that consists of several database actions carried out as a logical unit of data processing is called a transaction. One or more database operations, such as insert, remove, update, or retrieve data, are carried out throughout a transaction. There are two options for this atomic process: it is either carried out completely or not at all. A read-only transaction is one that solely involves the retrieval of data—no updates to the data are made.

Challenges in Distributed Transaction Control:

In distributed transaction control, controlling concurrency and isolation between several nodes is one of the main issues. Many transactions may be running concurrently in a distributed system and complex concurrency management procedures are needed to make sure they don't conflict with one another. Another difficulty is coordinating the atomicity of transactions across nodes since consistency requires a smooth commit or rollback synchronisation.

Because data is replicated among nodes, distributed transactions have complex consistency. Coordination becomes difficult when changes are made to one node and need to be propagated to others during distributed transactions. Failure management also becomes a crucial component, requiring strong systems to identify node failures, start recovery processes and guarantee that transactions impacted by failures are handled properly.

Achieving ACID Properties in Distributed Transactions:

Atomicity (A): Coordinating the commit or rollback of transactions across several nodes is necessary to achieve atomicity in distributed transactions. A commonly used technique for guaranteeing that every node either commits or aborts a transaction, the Two-Phase Commit (2PC) protocol lays the groundwork for distributed transactions to be atomic. Nevertheless, 2PC has a number of shortcomings, such as blocking problems and vulnerability to the "blocking coordinator problem."

Consistency (C):Maintaining a consistent state across all nodes following the completion of a transaction is the foundation of consistency in distributed transactions. Consistency in a distributed environment can be achieved in part by coordinating the update of distributed replicas and applying consistency constraints. Consistency is largely ensured by strategies using distributed naming services, replication methods and conflict resolution protocols.

Isolation (I): By separating the executions of concurrent transactions, isolation makes sure that they don't conflict with one another. Isolation in distributed transactions is managed with the use of strategies like locking mechanisms and multi-version concurrency control (MVCC). By preventing problems like dirty reads, non-repeatable reads and phantom reads, these procedures make sure that each transaction proceeds on its own.

Durability (D): Ensuring committed transactions continue even in the face of failures is a key component of durability. In order to achieve durability in remote databases, replication and logging systems are essential. Systems are able to recover from failures and guarantee

Notes

that committed changes are durable across distributed nodes by preserving transaction logs and replicating data across nodes.

Two-Phase Commit (2PC) Protocol: The Two-Phase Commit protocol is a crucial mechanism for achieving atomicity in distributed transactions. There are two primary stages to it:

Prepare Phase: All participating nodes receive a prepare message from the coordinator node during this phase, requesting them to vote on whether or not to commit the transaction. In response, each player votes to commit or abstain.

Commit Phase: In the commit phase, based on the votes collected during the prepare phase, the coordinator determines whether to commit or abort the transaction. The coordinator notifies each participant via commit message if all votes are in favour of committing. The coordinator notifies every participant of the abort vote if there is one.

Even though 2PC is widely used, it has limitations, such as blocking problems and coordinator failure susceptibility. Issues could emerge that affect the overall dependability and performance of distributed transactions, such as nodes waiting endlessly or the coordinator failing throughout the protocol.

Optimistic Concurrency Control: Unlike 2PC, optimistic concurrency control enables transactions to continue without first acquiring locks. This method postpones conflict resolution until the commit step, assuming that disputes are uncommon. Although optimistic concurrency control can improve concurrency, it may also cause more rollbacks in the event of a disagreement, which could negatively affect system performance as a whole.

Future Considerations:

Future developments in computing paradigms and upcoming technologies bring additional factors to the table for distributed transaction control.

- Blockchain and Smart Contracts: Blockchain technology provides decentralised consensus methods and smart contracts as viable options to provide distributed transaction control in a decentralised and trustless way. The immutability and transparency of blockchain technology can improve distributed transactions' dependability and integrity.
- Machine Learning for Dynamic Coordination: It seems promising to use machine learning methods to dynamically coordinate remote transactions. The efficiency of establishing ACID features may be improved by adaptive systems that are able to learn from and adjust to the peculiarities of the distributed environment. Machine learning models can analyse transaction patterns, identify possible conflicts and optimise coordination solutions dynamically.
- Edge Computing and IoT: Distributed transaction control is facing new issues with the emergence of edge computing and the Internet of Things (IoT). In these resource-constrained edge situations, transaction control optimisation calls for creative solutions. Because edge computing is decentralised, it requires solutions that can adapt to changing computational capacities, sporadic connectivity and dynamic data distribution.

3.1.9 Query Processing in Distributed Database

A DDBMS query may need information from databases spread across multiple sites. Relation databases with physically divided components are supported by some database systems. One relation may be divided into sub-relations and these subrelations disseminated to several sites, or distinct relations may live at different sites, or multiple

Notes

copies of a single relation may be spread among several sites. It could be required to move data across different sites in order to assess a query submitted at one location. As a result, it's critical to minimise the amount of time needed to access this type of query, as this will mostly consist of data transmission time between locations rather than computation or disc storage retrieval time.

Semi-JOIN

The transmission or communication cost in a distributed query processing system is significant. Consequently, the semijoin operation is used to lower the communication costs by reducing the size of a relation that needs to be communicated. As indicated in the figure below, let's assume that the relations R (EMPLOYEE) and S (PROJECT) are kept at sites C (Mumbai) and B (London), respectively. To create a project allocation list, a user queries site C, requesting the computation of the JOIN of the two relations provided as

JOIN (R, S)

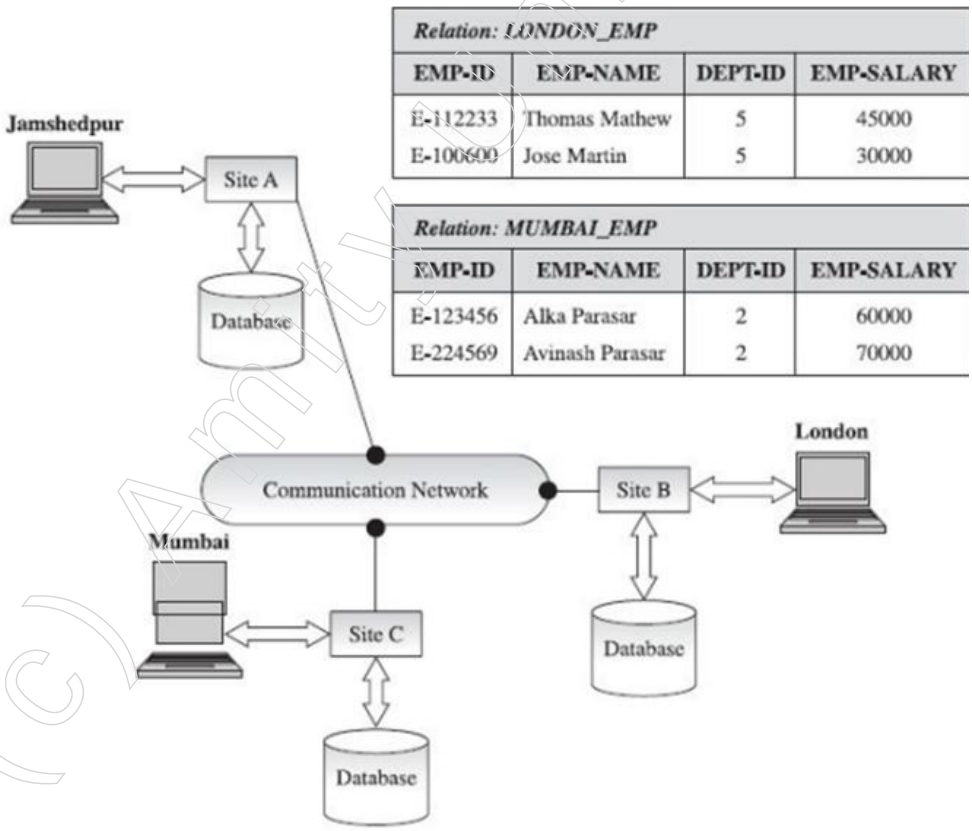
or JOIN (EMPLOYEE, PROJECT)

Sending all of connection S's (PROJECT) properties to site C in Mumbai and computing the JOIN there is one method of merging the aforementioned relations. This would have a high communication cost because it would need transmitting all 12 values of relation S (PROJECT).

An alternative method would be to project the relation C (PROJECT) at site B (London) first on the attribute EMP-ID, then send the result to site C (Mumbai). This can be calculated as follows:

X = ∏EMP-ID (S)

or X = ∏EMP-id (PROJECT)



Relation: LONDON_EMP

EMP-ID	EMP-NAME	DEPT-ID	EMP-SALARY
E-112233	Thomas Mathew	5	45000
E-100600	Jose Martin	5	30000

Relation: MUMBAI_EMP

EMP-ID	EMP-NAME	DEPT-ID	EMP-SALARY
E-123456	Alka Parasar	2	60000
E-224569	Avinash Parasar	2	70000

Notes

Relation: MUMBAI_EMP			
EMP-ID	EMP-NAME	DEPT-ID	EMP-SALARY
E-123456	Alka Parasar	2	60000
E-224569	Avinash Parasar	2	70000

Relation: LONDON_EMP			
EMP-ID	EMP-NAME	DEPT-ID	EMP-SALARY
E-112233	Thomas Mathew	5	45000
E-100600	Jose Martin	5	30000

Figure: An example of data replication

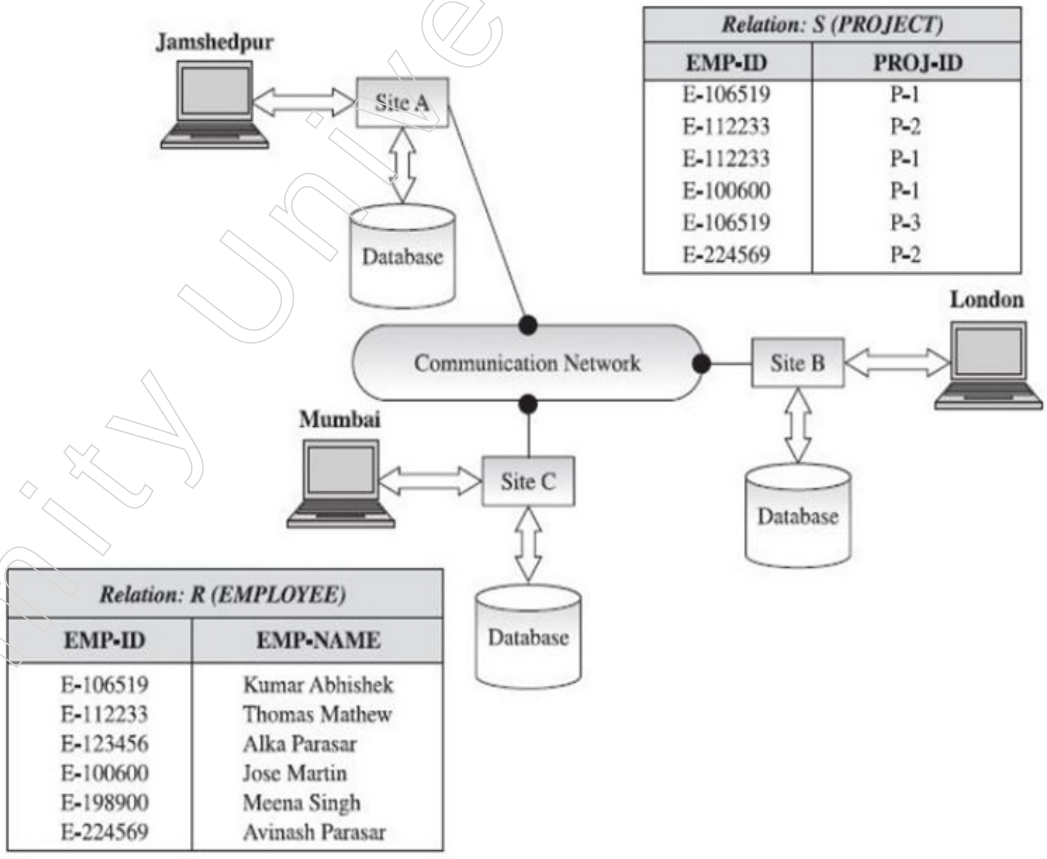
Image Source: Database Management System, S K Sinha, Second Edition

The figure below displays the projection operation's outcome. Now, at site C (Mumbai), the relation R (EMPLOYEE) tuples that share the same value for the attribute EMP-ID as a tuple in R are chosen X = ∏EMP-ID

(PROJECT) through a join and is calcifiable as

Y = JOIN (R, X)

or Y = EMPLOYEE |×| X



Relation: S (PROJECT)	
EMP-ID	PROJ-ID
E-106519	P-1
E-112233	P-2
E-112233	P-1
E-100600	P-1
E-106519	P-3
E-224569	P-2

Relation: R (EMPLOYEE)	
EMP-ID	EMP-NAME
E-106519	Kumar Abhishek
E-112233	Thomas Mathew
E-123456	Alka Parasar
E-100600	Jose Martin
E-198900	Meena Singh
E-224569	Avinash Parasar

Figure: Obtaining a join using semijoin

Image Source: Database Management System, S K Sinha, Second Edition

The term "semijoin" refers to the complete process of first projecting the relation S (PROJECT) and then executing the join; it is symbolised by |×|. This implies that

Notes

Y = EMPLOYEE |×| PROJECT

|×| EMPLOYEE |×| X

The semijoin operation's outcome is displayed in the figure below. However, as is evident, following the semijoin operation, the intended outcome is not achieved. The amount of tuples of relation R (EMPLOYEE) that must be communicated at site B (London) is decreased by the semijoin operation. As seen in the figure below, the final result is obtained by connecting the reduced relation R (EMPLOYEE) and relation S (PROJECT) and it may be computed as

R |×| S = Y |×| S or EMPLOYEE |×| PROJECT = Y |×| PROJECT

To cut down on communication costs, utilise the semijoin operator (|×|). Semijoin can be described as follows if Z is the outcome of relations R and S semijointly:

Z = R |×| S

The set of tuples from relation R that join with a tuple or tuples from relation S is represented by Z. Tuples of relation R that don't combine with any tuple in relation S are absent from Z. Z, then, stands for the decreased R that can be conveyed to a S site for joining with it. The size of Z would be a small percentage of the size of R if the join of R and S were very selective. We now join P with S to obtain the join of R and S and given as

T = Z |×| S = (R |×| S) |×| S

= (S |×| R) |×| R

= (R |×| S) |×| (S |×| R)

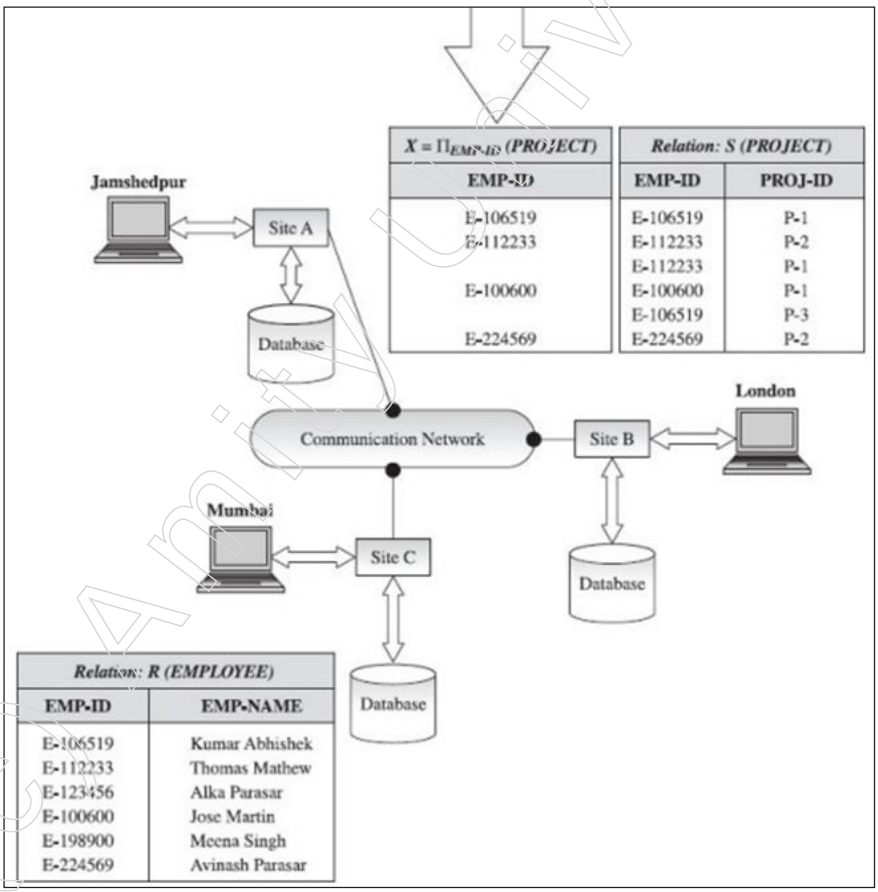


Figure: Result of projection operation at site B

Image Source: Database Management System, S K Sinha, Second Edition

Notes

R |×| S can be understood as R semijoin S, or as R reduced by S. The semijoin is a reduction operator. The semijoin operation is not associative, it should be noticed. This indicates that EMPLOYEE |×| PROJECT and PROJECT |×| EMPLOYEE are not the same in our example of the interactions between EMPLOYEE and PROJECT. The former results in fewer EMPLOYEE tuples, while the latter has the same relationship as PROJECT.

Summary

- A parallel database is a type of database system that harnesses the power of parallel processing by distributing data and query tasks across multiple processors or nodes. This architecture enables simultaneous and efficient execution of operations, reducing response times for complex queries and analytical tasks. Key features include data partitioning, parallel query execution, loading and indexing, contributing to enhanced scalability and fault tolerance.
- Distributed databases refer to systems where data is spread across multiple interconnected nodes or servers, allowing for efficient storage, processing and retrieval of information. In a distributed database, each node retains a portion of the data and the system coordinates operations across these nodes.
- Fragmentation is a database design strategy that aims to enhance efficiency, reduce data transfer overhead and improve system performance by breaking down a database into smaller, more manageable pieces based on specific criteria. Fragmentation is often employed in distributed database systems, where data is distributed across multiple nodes or servers. This distribution can lead to improved parallel processing and enhanced system performance.
- Implementing parallel databases requires a combination of software, hardware and architectural considerations to achieve optimal performance and scalability. The choice of parallelisation techniques and strategies depends on the specific requirements and characteristics of the database workload.
- Transaction control in distributed databases is a complex process that involves careful coordination, communication and recovery mechanisms to uphold the principles of ACID across multiple nodes. Implementing robust distributed transaction management is critical for maintaining data integrity and system reliability in distributed computing environments.
- Query processing in distributed databases is complex, requiring careful consideration of data distribution, parallel execution and coordination across nodes. Efficient strategies for decomposition, distribution and optimisation contribute to achieving optimal performance in distributed query processing systems.

Glossary

- DBMS: Database Management System
- I/O: Input/Output
- CPU: Central Processing Unit
- IoT: Internet of Things
- SQL: Structured Query Language
- DDBS: Distributed Database System
- DDL: Data Definition Language
- 2PC: Two-Phase Commit

- 3PC: Three-Phase Commit

Notes

Check Your Understanding

1. What is the primary goal of parallel databases?
   a) Minimising data redundancy b) Enhancing data security
   c) Improving data distribution d) Optimising data processing performance
2. How does horizontal fragmentation contribute to parallel database efficiency?
   a) Minimises data transfer b) Enhances data security
   c) Facilitates data replication d) Improves query readability
3. Which type of database architecture is often associated with parallel databases?
   a) Relational database b) Distributed database
   c) Hierarchical database d) Object-oriented database
4. In parallel databases, what is load balancing aimed at achieving?
   a) Minimising query complexity b) Distributing workload evenly
   c) Maximising data redundancy d) Reducing data distribution
5. What is a common fault-tolerance mechanism in parallel databases?
   a) Horizontal fragmentation b) Query decomposition
   c) Data replication d) Load balancing

Exercise

1. Define parallel database and distributed databases.
2. Define principles for distributed databases.
3. Define the architectures of parallel databases.
4. How to implement parallel databases?
5. Explain fragmentation.

Learning Activities

1. Consider a scenario where an organisation is planning to implement a new database system to handle its extensive data requirements. Discuss in detail the considerations and factors involved in choosing between a Parallel Database and a Distributed Database architecture. Provide a comprehensive analysis of the features, advantages and challenges associated with each approach.
2. Analyse the importance of query optimisation in a distributed database. Highlight specific techniques used to optimise queries, considering factors such as data distribution, indexing and parallel processing.

Check Your Understanding- Answers

1. d 2. a 3. b 4. b
5. c

Notes

Module - IV: Databases on the Web and Semi Structured Data

Learning Objectives

At the end of this module, you will be able to:

- Define web interfaces
- Discuss overview of XML
- Analyse structure of XML data
- Define document schema
- Understand querying XML data
- Discuss storage of XML data
- Understand XML Applications
- Define semi structured data model

Introduction

The manner that information is stored, accessed and shared has been completely transformed by the incorporation of databases into the World Wide Web. Databases are essential to the field of web development and data management because they enable dynamic, data-driven applications. These applications include social networking networks, e-commerce platforms, content management systems and more.

The idea of “databases on the web,” wherein conventional database systems are expanded and modified to address the particular difficulties brought about by the distributed and networked nature of the internet, lies at the core of this evolution. The foundation of data storage for online applications is made up of NoSQL databases like MongoDB and Cassandra and Relational Database Management Systems (RDBMS) like MySQL, PostgreSQL and Oracle. These systems guarantee the dependability and durability of vital data by providing scalable and effective solutions for managing structured data.

Simultaneously, web-based databases now have a new dimension due to the growth of semi-structured data. Semi-structured data is a format that can be used to represent a variety of information kinds since it combines elements of both structured and unstructured data. XML (eXtensible Markup Language) is a crucial semi-structured data format that facilitates the hierarchical organisation of data via user-defined tags. XML has established itself as a web standard for data transfer, allowing applications, platforms and systems to communicate with one other seamlessly.

Another well-known semi-structured data format that is becoming more and more popular is JSON (JavaScript Object Notation). This is because it is easy to use and works with JavaScript. Because of its simple syntax, JSON is especially well-suited for online applications, where data interchange between the client and server is frequently done.

The introduction of online services, such as SOAP (Simple Object Access Protocol) and RESTful APIs, highlights the significance of databases on the internet even more. By utilising the web’s fundamental concepts, these services facilitate data interchange between various applications, offering a standardised and compatible mode of communication. Specifically, RESTful APIs have emerged as a key component of loosely linked, scalable web architectures that facilitate the smooth integration of databases with online applications.

Notes

The key issues to keep in mind when navigating the dynamic intersection of databases and the web are security, performance and scalability. To safeguard sensitive data from potential dangers, security measures are essential. These methods include encryption techniques, secure transmission channels and strong authentication systems. Caching techniques and database indexing are examples of performance optimisations that make sure web applications continue to provide responsive user experiences even with increasing data quantities. Scalability, achieved through techniques like sharding and replication, allows databases to accommodate increasing workloads and user demands.

The development of online databases and the acceptance of semi-structured data in this context signify a paradigm change in the way we conceptualise, handle and use information in the digital age. The synergy between traditional database systems, semi-structured data formats and web technologies empowers developers to construct powerful, adaptive and interconnected applications that shape the way we interact with and harness the power of data on the World Wide Web.

4.1 XML and Its Applications

Extensible Markup Language (XML) applications were not intended for use with databases. Actually, XML has its roots in document management and is evolved from a language for structuring huge documents known as the Standard Generalised Markup Language (SGML), much like the Hyper-Text Markup Language (HTML) on which the World Wide Web is built. XML, on the other hand, is meant to represent data, unlike SGML and HTML. When an application needs to combine data from multiple apps or communicate with another program, it is very helpful as a data format. Numerous database problems, such as those with the organisation, manipulation and querying of the XML data, occur when XML is utilised in these situations. This session introduces XML and covers the interchange of data in XML document format as well as the management of XML data using database approaches.

XML stands for Extensible Markup Language and because of its structure, ease of use and readability, it is a vital tool in many different fields. Although XML was primarily created for data transfer, it has many uses. XML is frequently used in web development to represent and exchange data between a client and a server, enabling smooth communication in applications such as online shopping platforms and content management systems. Furthermore, configuration files frequently contain XML, which makes it possible to define and modify program and system settings. Databases frequently use XML to organise and structure data during data storage and retrieval, facilitating effective data management and interoperability. Web services are another important use for XML-based protocols, such as SOAP (Simple Object Access Protocol), which are used to communicate between various platforms and systems. Furthermore, XML is essential for representing documents with intricate structures, like scientific articles or contracts, because of its hierarchical nature, which improves readability and organisation. All things considered, XML's adaptability makes it a crucial component of contemporary computing, supporting a wide range of applications and facilitating the smooth interchange and representation of structured data.

4.1.1 Web interfaces

Web interfaces are essential for dealing with databases because they give users the ability to enter, retrieve and modify data via a web browser. This contact is assisted by web apps that use computer languages like HTML, CSS and JavaScript to construct dynamic and user-friendly interfaces. Web interfaces and databases can be integrated to facilitate easy data administration and retrieval for a variety of applications, including e-commerce

Notes

platforms and content management systems. Databases are repositories of organised data and the effectiveness of web applications depends on the effective administration of these systems. By encapsulating the complexity of database queries and processes, web interfaces and databases enable people to engage with data in an intuitive way.

While static Web pages with fixed text and other objects can be generated using basic HTML, the majority of e-commerce systems require Web sites that have interactive features and employ user input to choose specific data from a database for display. Because the data gathered and shown on these pages varies based on user input, they are referred to as dynamic webpages. A banking app, for instance, would first obtain the user's account number before retrieving the account amount from the database and displaying it. We talked about creating dynamic Web pages for apps using scripting languages like PHP. When required by the applications, XML can be used to transfer data in self-describing text files between different programs running on different computers.

Web applications use front-end technologies to create their user interface. Web pages are structured by HTML, presented and styled by CSS and dynamic behaviour and interaction are added by JavaScript. React, Angular and Vue.js are examples of contemporary front-end frameworks and tools that facilitate the creation of intricate and responsive online interfaces. Effective web interface design heavily relies on User Experience (UX) and User Interface (UI) design principles. The utilisation of responsive design guarantees cross-platform compatibility, offering a smooth and uninterrupted experience across PCs, tablets and smartphones.

Web interfaces are offered by numerous Internet applications to access data kept in one or more databases. A common term for these databases is "data sources." For Internet applications, three-tier client/server designs are frequently used. Web interfaces that show Web pages on desktops, laptops and mobile devices are how Internet database applications communicate with users. Using hypertext documents is a typical way to specify the formatting and content of Web pages. These documents can be written in a number of languages, the most popular of which is HTML (HyperText Markup Language). While HTML is a common tool for organising and displaying Web pages, it is not appropriate for describing structured data that is taken out of databases. Extensible Markup Language, or XML, is a new language that has become the industry standard for organising and transferring data in text files over the Internet. JSON is an additional language that can be utilised for the same objective.

Instead of only defining the syntax of a Web page for display on a screen, XML can also give information about the organisation and meaning of the contents within the page. XML and JSON documents are referred to as self-describing documents because they provide descriptive data in a text file, such as attribute names and values. Web pages' formatting elements are provided independently, for instance, through the use of transformation languages like XSLT (Extensible Stylesheet Language for Transformations, also known as XSL Transformations) or formatting languages like XSL (Extensible Stylesheet Language). A few experimental database systems based on XML have been constructed thus far, but XML has also been suggested recently as a potential model for data retrieval and storage.

Back-End Technologies and Database

Web applications' back ends manage server-side functions, such as database interactions. Request processing, database queries and dynamic content generation are frequently carried out using server-side scripting languages such as PHP, Python, Ruby and Node.js. Back-end development is made easier by server-side frameworks like Express.

Notes

js, Flask, Ruby on Rails and Django. Relational databases (like MySQL and PostgreSQL), NoSQL databases (like MongoDB and Cassandra) and in-memory databases (like Redis) are some examples of the different types of databases. Performance needs, scalability and data structure all play a role in the database selection.

Database Connectivity and Interaction

Database Management System (DBMS) drivers or Object-Relational Mapping (ORM) frameworks are used by web applications to communicate with databases. SQL queries can be executed directly from the database thanks to DBMS drivers. Conversely, ORM frameworks abstract the underlying database interactions by mapping database entities to objects in the application code.

Database interaction is based on the CRUD (Create, Read, Update, Delete) activities. Through forms, buttons and other UI elements, web interfaces provide these functions, enabling users to enter data, get information and edit database records.

Security Considerations

Web applications must prioritise security, particularly when handling sensitive data kept in databases. Common vulnerabilities such as SQL injection can be avoided with the use of security measures like parameterised queries, input validation and authentication methods.

Secure socket layers (SSL), appropriate user authentication procedures and encryption methods are essential for protecting data when it is being transmitted between the database and the online interface. By ensuring users have the proper authorisation, role-based access control helps prevent unwanted access to private data.

RESTful APIs and Web Services

Web services or Representational State Transfer (RESTful) APIs are frequently used by web interfaces to connect to databases. These APIs enable for the smooth integration of web interfaces and databases by defining a set of standard operations (GET, POST, PUT and DELETE) that may be executed on resources.

With a scalable and loosely linked design offered by RESTful APIs, various system components can develop separately. They are frequently employed in web applications to transfer data between the front and back ends.

Binding and Front-End Frameworks

Data binding is a mechanism used by front-end frameworks such as React, Angular and Vue.js to synchronise data between the database and the web interface. A real-time and dynamic user experience is provided by two-way data binding, which makes sure that changes made in the UI are reflected in the database and vice versa.

Additionally, front-end frameworks make it easier to create Single Page Applications (SPAs), which improve responsiveness and minimise the need for frequent page reloads by running the entire program on a single web page.

New trends and technologies are constantly influencing the creation of database interactions and web interfaces as technology advances. PWAs, or progressive web apps, provide improved offline functionality and app-like experiences. An technique to data retrieval that is more adaptable and effective than REST is GraphQL.

With the introduction of serverless architecture, web application deployment and scalability have become easier, freeing developers to concentrate on writing code rather

Notes

than maintaining server infrastructure. By dividing larger programs into more manageable, independent services that can interact with one another, microservices architecture encourages modular development.

4.1.2 Overview of XML

The World Wide Web Consortium (W3C) created XML, or extensible markup language, a flexible and extensively used markup language. XML is a text-based format that was first introduced in 1998 and is used to store and transfer data in a standardised, legible way. One of its main characteristics is that it uses a tag-based syntax, in which elements can be nested inside one another and are defined by the opening and closing tags that surround content. A single root element is required for XML documents and elements may contain attributes that provide more information.

XML is frequently used to enable data exchange across many platforms, systems and programming languages. It offers a standardised method for arranging and representing a variety of data kinds, including documents, online services, configuration files and data interchange formats. To verify adherence to a predetermined structure, XML documents can be verified against Document Type Definitions (DTDs) or XML Schema Definitions (XSDs). Because of its readability, flexibility and ease of use, XML is widely used in many different fields and is regarded as a foundational technology for data interchange and representation.

Fundamentally, XML is a text-based format created to transfer and store data in a manner that is both machine- and human-readable. In contrast to HTML, which is mostly used for presenting information on the web, XML is concentrated on describing and organising data. Because of its adaptability, it is a well-liked option for many applications, such as online services, configuration files and data interchange formats.

The essential building blocks of an XML document are called elements. Elements are defined using tags surrounded in angle brackets (<>). An XML element typically consists of an opening tag, a content tag and a closing tag. It is possible to nest elements inside of one another to create a hierarchical structure.

Attributes are defined inside the opening tag and offer further details about elements. They are helpful for adding metadata to elements and are shown as name-value pairs. As an illustration:

```
<book category=”fiction”>
<title>XML Basics</title>
<author>Jane Smith</author>
</book>
```

Here, “book” is an element with the attribute “category.”

Elements. In an XML document, elements—also referred to as tags—are the fundamental building components. An element ELM’s content is identified at the beginning with, also known as the start tag and at the conclusion with, also known as the end tag. All of the information in the sample document is contained in the BOOKLIST element in our example document. All information pertaining to a particular book is denoted by the element BOOK. XML elements are case sensitive: the element is distinct from . The elements need to be nested correctly. Any start tag that shows up inside another tag’s content needs to have an end tag that matches. Take the following XML snippet, for instance:

Notes

```
<BOOK>
    <AUTHOR>
     <FIRSTNAME>Richard</FIRSTNAME>
     <LASTNAME>Feynman</LASTNAME>
    </AUTHOR>
</BOOK>
```

The components LASTNAME and FIRSTNAME are nested inside the element AUTHOR, which is fully nested inside the element BOOK.

Attributes. Descriptive characteristics are properties that give more details about an element. The start tag of an element contains the attribute values. Let ELM, for instance, stand for an element that has the property att. We can set the value of att to value with the following expression: . Quotations must enclose all attribute values. The element BOOK in the code below has two attributes. The book's genre (fiction or scientific) is indicated by the attribute genre, while the format (hardcover or paperback) is indicated by the attribute format.

```
<?XML version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE BOOKLIST SYSTEM "books.dtd">
<BOOKLIST>
    <BOOK genre="Science" format="Hardcover">
     <AUTHOR>
          <FIRSTNAME>Richard</FIRSTNAME>
          <LASTNAME>Feynman</LASTNAME>
     </AUTHOR>
     <TITLE>The Character of Physical Law</TITLE>
     <PUBLISHED>1980</PUBLISHED>
    </BOOK>
    <BOOK genre="Fiction">
     <AUTHOR>
          <FIRSTNAME>R.K.</FIRSTNAME>
          <LASTNAME>Narayan</LASTNAME>
     </AUTHOR>
     <TITLE>Waiting for the Mahatma</TITLE>
     <PUBLISHED>1981</PUBLISHED>
    </BOOK>
    <BOOK genre="Fiction">
     <AUTHOR>
          <FIRSTNAME>R.K.</FIRSTNAME>
          <LASTNAME>Narayan</LASTNAME>
     </AUTHOR>
     <TITLE>The English Teacher</TITLE>
     <PUBLISHED>1980</PUBLISHED>
    </BOOK>
</BOOKLIST>
```

Notes

Entity references. We refer to the use of an entity in an XML document as an entity reference. Entities are shortcuts for sections of common text or the content of external files. Anywhere in the document where an entity reference exists, its content takes its place textually. References to entities begin with a "&" and conclude with a ";". Five predefined XML entities serve as placeholders for characters in the language that have unique meanings. For instance, the entity lt must represent the reserved < character, which appears at the start of an XML command. The remaining four reserved characters are &, >, " and ' and they are represented by the entities amp, gt, quot and apos. For instance, in an XML document, the sentence "1 < 5" needs to be encoded as '1<5&apos. Additionally, we may add any Unicode character to the text by using entities. Similar to ASCII, Unicode is a standard for character representations. For instance, we can use the entity reference &#x3042 to display the Japanese Hiragana character "a."

Comments. We can insert comments anywhere in an XML document. Comments start with. Comments can contain arbitrary text except the string.

Document type declarations (DTDs). In XML, we can define our own markup language. A DTD is a collection of guidelines that enables us to define our own collection of components, properties and entities. As a result, a DTD is essentially a grammar that specifies which tags are permitted, how they can be nested and in what order they can occur.

- The XML declaration opens the document. An example of an XML declaration is the first line of the XML document displayed in above code.
- Every other element is contained in a root element. The element BOOKLIST is the root element in our sample.
- Every piece needs to be correctly nested. According to this requirement, an element's start and end tags have to occur inside the same enclosing element.
- All other elements in the XML document must be contained within a single root element. The document's top-level structure is formed by this root element.

For an XML document to be deemed well-formed, it must follow specific syntactic guidelines. This entails having a single root element, balanced tags and appropriate nesting. To guarantee its integrity, the document's structure is examined in comparison to these guidelines. An XML document can be checked for validity against an XML Schema Definition (XSD) or a Document Type Definition (DTD) in addition to being well-formed. The structure and data types anticipated in the XML document are specified by these validation techniques. By guaranteeing that the document complies with a predetermined schema, validation raises the bar for data integrity and interoperability.

The usage of elements and attributes from various XML vocabularies in the same document without creating conflicts is made possible by XML Namespaces. This is necessary to prevent naming collisions or to combine data from several sources. The root element of an XML document has a namespace prefix, which is used to declare namespaces and then qualify items inside the document. As an illustration:

```
<root xmlns:ns="http://www.example.com/ns">
    <ns:element1>Content 1</ns:element1>
    <ns:element2>Content 2</ns:element2>
</root>
```

One of the most potent languages for converting XML documents into other formats is called Extensible Stylesheet Language Transformations (XSLT). To specify how the input XML should be changed into the intended output, which can be another XML document,

HTML, or even plain text, XSLT uses a set of rules and templates. XSLT promotes flexibility and reusability by enabling developers to isolate the presentation of data from its underlying structure. It is frequently used for activities like changing XML documents to satisfy specifications or translating XML data into HTML for online display.

Notes

Communication between multiple systems and platforms. Web services facilitate interoperability in dispersed computing systems by using XML to format communications sent back and forth between clients and servers. A protocol called Simple Object Access Protocol (SOAP) uses XML extensively to structure messages. The functionality provided by a web service is described, together with the ways in which clients can interact with it, using XML-based languages such as Web Services Description Language (WSDL).

4.1.3 Structure of XML data

The element is the basic building block of an XML document. All text that appears between a pair of matching start and end tags is considered an element. All other elements in an XML document must be contained within a single root element. The element is the root element in the code example below. An XML document's elements also need to nest correctly.

```
<purchase order>
<identifier> P-101 </identifier>
<purchaser>
<name> Cray Z. Coyote </name>
<address> Mesa Flats, Route 66, Arizona 12345, USA </address>
</purchaser>
<supplier>
<name> Acme Supplies </name>
<address> 1 Broadway, New York, NY, USA </address>
</supplier>
<itemlist>
<item>
<identifier> RS1 </identifier>
<description> Atom powered rocket sled </description>
<quantity> 2 </quantity>
<price> 199.95 </price>
</item>
<item>
<identifier> SG2 </identifier>
<description> Superb glue </description>
<quantity> 1 </quantity>
<unit-of-measure> liter </unit-of-measure>
<price> 29.95 </price>
</item>
</itemlist>
<total cost> 429.85 </total cost>
<payment terms> Cash-on-delivery </payment terms>
<shipping mode> 1-second-delivery </shipping mode>
</purchaseorder>
```

Notes

For instance:

<course>...<title>...</title>...</course>

is properly nested, whereas:

<course>...<title>...</course>...</title>

is not properly nested.

Even while appropriate nesting makes sense intuitively, there are formal definitions we can use. If text appears between an element's starttag and endtag, it is considered to be in the element's context. If each start tag has a distinct matching end tag within the same parent element, then tags are correctly nested.

Take note that, as in the code below, text can be used with an element's subelements. Similar to multiple other characteristics of XML, this liberty is more appropriate for document processing than for data processing and thus is not very helpful for XML representation of more structured data, like database information.

...

<course>

This course is being offered for the first time in 2009.

<course id> BIO-399

</course>

<title> Computational Biology </title>

<dept name> Biology

</dept>

<credits> 3 </credits>

</course>

...

An additional method of representing information is offered by the capability to nest items inside other elements. The code below represents a portion of the university data, except it has departmental items nested inside course elements. Finding every course that a department offers is made simple by the nested representation. In a similar vein, course identifiers assigned by an instructor are nestled inside teacher elements. There would be several course id elements within the associated instructor element if an instructor taught more than one course. Details of teachers Brandt and Crick are omitted from code for lack of space, but are similar in structure to that for Srinivasan.

```
<university-1>
    <department>
     <dept name> Comp. Sci.
    </dept>
    <building> Taylor </building>
    <budget> 100000 </budget>
    <course>
     <course id> CS-101
    </course>
```

Notes

```
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    <course>
        <course id> CS-347
    </course>
    <title> Database System Concepts </title>
    <credits> 3 </credits>
    </course>
    </department>
    <department>
        <dept name> Biology
    </dept>
    <building> Watson </building>
    <budget> 90000 </budget>
```

```
    <course>
        <course id> BIO-301
    </course>
    <title> Genetics </title>
    <credits> 4 </credits>
    </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept name> Comp. Sci.
    </dept>
    <salary> 65000. </salary>
    <course id> CS-101
    </coursr>
    </instructor>
    </university-1>
```

While hierarchical representations are inherent in XML, they could result in unnecessary data storage. As an illustration, let's say that the instructor element stores course details nested inside it, as the code below illustrates. Course details including the title, department and credits would be redundantly kept with each instructor connected to the course if many instructors were teaching it.

Notes

```
<university-2>
    <instructor>
    <ID> 10101 </ID>
    <name> Srinivasan </name>
    <dept name> Comp. Sci.< dept name>
    <salary> 65000 </salary>
    <teaches>
    <course>
    <course id> CS-101
    </course>
    <title> Intro. to Computer Science </title>
    <dept name> Comp. Sci.
    </dept>
    <credits> 4 </credits>
    </course>
</teaches>
</instructor>
<instructor>
    <ID> 83821 </ID>
    <name> Brandt </name>
    <dept name> Comp. Sci.< dept name>
    <salary> 92000 </salary>
    <teaches>
<course>
    <course id> CS-101
    </course>
    <title> Intro. to Computer Science </title>
    <dept name> Comp. Sci.
</dept>
    <credits> 4 </credits>
</course>
</teaches>
</instructor>
</university-2>
```

In order to prevent joins, nested representations are frequently employed in XML data transfer systems. In contrast to a normalised representation, which might necessitate joining buy order records with a corporate address relation in order to obtain address

Notes

information, a purchase order would keep the sender and recipient's whole address redundantly on many purchase orders.

Apart from elements, XML also defines the concept of an attribute. For example, the code below illustrates how a course's course identifier might be expressed as an attribute. An element's attributes are shown as name=value pairs preceding a tag's closing ">". Attributes are just strings devoid of any markup. In addition, unlike subelements, which can exist more than once in a particular tag, attributes can only appear once.

```
...
<course course id="CS-101">
<title> Intro. to Computer Science</title>
<dept name> Comp. Sci.
</dept>
<credits> 4 </credits>
</course>
...
```

Keep in mind that the distinction between an attribute and a subelement is crucial when creating documents since an attribute is just text that is not visible in the final product, whether it is printed or displayed. Nevertheless, this distinction is less important in database and data sharing uses of XML and the decision to represent data as an attribute or a subelement is often discretionary. Generally speaking, it's best to store all additional data in subelements and only utilise attributes to represent identifiers.

Finally, a grammatical note: an element of the form can be shortened as follows if it does not contain any text or subelements; the shortened elements may still have attributes. A namespace method has been created to enable organisations to designate globally unique names to be used as element tags in documents, as XML documents are intended to be shared throughout applications. A universal resource identifier (such as a Web URL) is appended to each tag or property in a namespace. Yale University, for instance, might prepend a unique identification with a colon to each tag name in order to guarantee that the XML documents it developed wouldn't duplicate tags used by any business partner's XML documents. The university might make use of a URL on the Web like:

https://www.amity.edu/

as a distinct identity. Since it would be cumbersome to use lengthy unique identities in each tag, the namespace standard offers a mechanism to create identifier abbreviations.

The root element (university) in the code below has an attribute called xmlns:amity, which indicates that amity is specified as an acronym for the URL mentioned above. The graphic shows how the abbreviation can be used in different element tags. Multiple namespaces that are declared as part of the root element are allowed in a document. Then, distinct components can be linked to distinct namespaces.

Using the attribute xmlns in place of xmlns: amity in the root element will define a default namespace. The default namespace would thus apply to elements lacking an explicit namespace prefix. Occasionally, we must store data that contain tags without allowing the tags to be read as XML tags. To enable this, XML allows the following construct:

Notes

```
<![CDATA[<course> …</course>]]><university xmlns:amity="https://www.amity.edu"> ...
<amity:course>
    <amity:course id> CS-101
</amity:course>
<amity:title> Intro. to Computer Science</amity:title>
<amity:dept name> Comp. Sci.
</amity:dept>
<amity:credits> 4 </amity:credits>
</amity:course>
...
</university>
```

The text is not interpreted as a tag, but rather as regular text data because it is encased in CDATA. Character data is referred to as CDATA.

4.1.4 Document Schema

Schemas are used in databases to limit the sorts of data that can be put there as well as what can be stored in the database. On the other hand, XML documents can be constructed by default without any associated schema, in which case an element can have any attribute or subelement. Given that XML documents are self-describing, this freedom might occasionally be acceptable, but it is usually not helpful when processing XML documents automatically as part of an application or even when formatting huge volumes of linked data in XML.

An XML document may include the document type definition (DTD), which is optional. Similar to a schema, a DTD's primary goal is to type and constrain the data that is contained in the document. But in reality, types are not constrained by the DTD in the sense of primitive kinds like string or integer. Rather, it restricts only an element's attribute and subelement appearance. The DTD consists mostly of a set of guidelines for the possible arrangement of subelements within of an element. A portion of an example DTD for a university information document is displayed in the code below;

```
<!DOCTYPE university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept name, building, budget)>
    <!ELEMENT course ( course id, title, dept name, credits)>
    <!ELEMENT instructor (IID, name, dept name, salary)>
    <!ELEMENT teaches (IID, course id)>
    <!ELEMENT dept name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
```

```
        <!ELEMENT IID( #PCDATA )>
        <!ELEMENT name( #PCDATA )>
        <!ELEMENT salary( #PCDATA )>
]>
```

Notes

Every declaration for an element's subelements takes the form of a regular expression. As a result, a university element in the DTD in the code above comprises of one or more components related to courses, departments, or instructors; the | operator indicates "or," while the + operator specifies "one or more." The? operator is used to provide an optional element (i.e., "zero or one"), but the operator is used to specify "zero or more," albeit it is not demonstrated here.

The subelements course id, title, department name and credits are contained in the course element (in that sequence). The DTD defines the attributes of the department and instructor's relational structure as subelements.

Finally, the components course id, title, dept name, credits, building, budget, IID, name and salary are all declared to be of type #PCDATA. Text data is indicated by the hashtag #PCDATA, which historically got its name from "parsed character data." Two further special type declarations are empty, which indicates that the element is empty and any, which indicates that the element's subelements are unconstrained; in other words, any element, including ones not specified in the DTD, may occur as a subelement of the element. An element's lack of a declaration is the same as specifically designating the type as any.

The DTD also declares the permitted characteristics for every element. An attributes' order is not enforced, in contrast to subelements. Types of attributes: CDATA, ID, IDREF and IDREFS. Type CDATA indicates simply that the attribute includes character data; the other three are more complicated and will be covered in more depth shortly. For example, the DTD line that follows indicates that element course has an attribute of type course id and that this attribute needs to have a value:

<!ATTLIST course course id CDATA #REQUIRED>

Both a type declaration and a default declaration are required for attributes. The default declaration can have one of two values: #IMPLIED, which indicates that no default value has been provided and the attribute may be omitted from the document, or #REQUIRED, which indicates that a value must be specified for the attribute in each element. If an attribute has a default value, the default value is automatically filled in when the XML document is read for each element that does not specify a value for the attribute.

A value that appears in an element's ID property cannot appear in any other element in the same document. An attribute of type ID gives the element a unique identification. An element may have no more than one attribute of type ID. (To avoid confusion with the type ID, we renamed the instructor relation's attribute ID to IID in the XML form.) A value that appears in the ID attribute of some element in the document must be present in an attribute of type IDREF, which is a reference to an element. A list of references, separated by spaces, is supported by the type IDREFS.

The sample DTD in the code below represents the identifiers of the course, department and instructor as ID attributes and the relationships between them as IDREF and IDREFS attributes. Course id has been made an attribute of course rather than a subelement in order to serve as the identifier attribute for the course elements. Every course element additionally has an IDREFS property teachers, which identifies the instructors that teach the course and an IDREF of the department that corresponds to the course. The instructor

Notes

elements contain an identifier attribute called IID and an IDREF attribute dept name specifying the department to which the instructor belongs. The department elements have an identifier attribute called dept name.

```
<!DOCTYPE university-3 [
    <!ELEMENT university ( (department|course|instructor)+)>
    <!ELEMENT department ( building, budget )>
    <!ATTLIST department
dept name ID #REQUIRED >
    <!ELEMENT course (title, credits )>
    <!ATTLIST course
course id ID #REQUIRED
dept name IDREF #REQUIRED
instructors IDREFS #IMPLIED >
    <!ELEMENT instructor ( name, salary )>
    <!ATTLIST instructor
IID ID #REQUIRED >>
⋯ declarations for title, credits, building,
budget, name and salary ⋯
] >
```

An example XML document based on the DTD in the code above is displayed below.

```
<university-3>
    <department dept name="Comp. Sci.">
    <building> Taylor </building>
    <budget> 100000 </budget>
</department>
<department dept name="Biology">
<building> Watson </building>
<budget> 90000 </budget>
</department>
<course course id="CS-101" dept name="Comp. Sci" instructors="10101 83821">
<title> Intro. to Computer Science </title>
<credits> 4 </credits>
</course>
<course course id="BIO-301" dept name="Biology" instructors="76766">
<title> Genetics </title>
<credits> 4 </credits>
</course>
<instructor IID="10101" dept name="Comp. Sci.">
<name> Srinivasan </name>
```

Notes

```
<salary> 65000 </salary>
</instructor>
<instructor IID="83821" dept name="Comp. Sci.">
<name> Brandt </name>
<salary> 72000 </salary>
</instructor>
<instructor IID="76766" dept name="Biology">
<name> Crick </name>
<salary> 72000 </salary>
</instructor>
</university-3>
```

In object-oriented and object-relational databases, the ID and IDREF properties play the same function as reference mechanisms, enabling the creation of intricate data associations. Document type definitions have a close relationship to XML's legacy in document formatting. For this reason, they are not appropriate in many respects to serve as the XML type structure for data-processing systems. Nevertheless, because they were a part of the original standard, several data exchange formats have been specified in terms of DTDs. The following are a few drawbacks of DTDs as a schema mechanism:

- It is not possible to type individual text components or properties any further. For example, it is not possible to confine the element balance to a positive value. Applications used for data processing and exchange are hampered by the absence of these restrictions since they need to include code to confirm the kinds of components and attributes.
- Using the DTD technique to express unordered sets of subelements is challenging. For data communication, order is rarely important (unlike text layout, where it is crucial). While the combination of alternation (the | operation) with the or the + operation as in above code provides the declaration of unordered collections of tags, it is considerably more difficult to define that each tag may only occur once.
- Nothing is typed in for IDs or IDREFSs. Consequently, it is impossible to define what kind of element an IDREFS or IDREF property should refer to. Consequently, even though this is absurd, the DTD in the code above does not stop a course element's "dept name" attribute from referring to other courses.

XML Schema

A more complex schema language called XML Schema was created in an attempt to address the shortcomings of the DTD system. After giving a quick introduction to XML Schema, we outline some of the ways that it enhances DTDs. Numerous built-in types, including string, integer, decimal date and boolean, are defined in the XML Schema. Furthermore, it supports user-defined types, which can be complicated kinds built with constructors like complexType and sequence, or simple types with extra constraints.

The codes below demonstrate how XML Schema can represent the DTD in the code above; we go over the XML Schema characteristics that the figures depict later. The first thing to notice is that XML Schema definitions use a range of tags provided by the schema to specify themselves in XML syntax. We prefix the XML Schema tag with the namespace prefix "xs:" in order to prevent conflicts with user-defined tags. The xmlns:xs specification in the root element associates this prefix with the XML Schema namespace:

Notes

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

Remember that xs can be replaced with any namespace prefix, so we could change all instances of "xs:" in the schema definition to "xsd:" without altering the meaning of the definition. This namespace prefix needs to come before any types that are defined by XML Schema.

The root element, university, is the initial element. UniversityType, the type for which is defined, is declared subsequently. The sorts of elements department, course, teacher and teaches are then defined in the example. Keep in mind that the type specification is contained in the body of each element with tag xs:element, which specifies each of these.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType"/>
<xs:element name="department">
<xs:complexType>
    <xs:sequence>
     <xs:element name="dept name" type="xs:string"/>
     <xs:element name="building" type="xs:string"/>
    <xs:element name="budget" type="xs:decimal"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="course">
<xs:element name="course id" type="xs:string"/>
<xs:element name="title" type="xs:string"/>
<xs:element name="dept name" type="xs:string"/>
<xs:element name="credits" type="xs:decimal"/>
</xs:element>
<xs:element name="instructor">
<xs:complexType>
    <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
```

The department type is described as complicated and it is further indicated that it consists of the following elements: budget, building name and department name. A complicated type has to be stated if it contains either attributes or nested subelements.

Notes

```
<xs:element name="teaches">
<xs:complexType>
    <xs:sequence>
    <xs:element name="IID" type="xs:string"/>
    <xs:element name="course id" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:complexType name="UniversityType">
<xs:sequence>
    <xs:element ref="department" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="course" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="instructor" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="teaches" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

As an alternative, the attribute type can specify the type of an element to be a predetermined type; note how the XML Schema types xs:string and xs:decimal are used to restrict the types of data components like credits and department name. The example concludes by defining the type UniversityType as having zero or more instances of the department, course, instructor and instructs. Take note of how an element that was previously defined is specified by using ref. With minOccurs and maxOccurs, an XML Schema can specify the lowest and maximum number of times a subelement appears. To allow zero or more department, course, instructor and teaches items, these must be explicitly defined as the default value for both minimum and maximum occurrences is 1.

The xs:attribute tag is used to specify attributes. For instance, we might have included the dept name to the attribute definition by

```
<xs:attribute name = "dept name"/>
```

inside the department element's declaration. While the default value of use is optional, adding the attribute use = "required" to the above attribute specification indicates that the attribute must be supplied. Even if elements are nested inside a sequence specification, attribute specifications would display right under the enclosing complexType definition. The syntax for creating named complex types with the xs:complexType element is the same as it is in the code above, with the exception that we add an attribute name = typeName to the xs:complexType element, where typeName is the name we want to give the type. As with xs:decimal and xs:string in our example, we can then use the named type to specify the type of an element using the type attribute.

A relational schema permits the declaration of constraints in addition to types. Keys and key references can be specified in XML Schema, which is equivalent to the SQL definition of primary and foreign keys. A unique constraint, also known as a primary-key constraint in SQL, makes sure that the attribute values don't repeat inside the relation.

Notes

In the context of XML, we must define a scope that contains unique values that make up a key. Field declarations list the components or attributes that make up the key and the selector, which is a path expression that defines the constraint's scope, specifies these items. We add the following constraint specification to the schema definition to ensure that the department name constitutes a key for department items beneath the parent university element:

<xs:key name="deptKey">

<xs:selector xpath="/university/department"/>

<xs:field xpath="dept name"/>

</xs:key>

Accordingly, the following definition of a foreign-key constraint from course to department might be applied:

<xs: name="courseDeptFKey" refer="deptKey">

<xs:selector xpath="/university/course"/>

<xs:field xpath="dept name"/>

</xs:keyref>

It should be noted that the field specification indicates the referring attributes, whereas the refer attribute provides the name of the key declaration being referenced. XML Schema is extensively used nowadays and has various advantages over DTDs. The following are a few advantages we have observed in the aforementioned examples:

- It enables the text that appears in elements to be restricted to particular types, including complicated types like sequences of elements of different types or numeric types in specific forms.
- It makes it possible to build user-defined kinds.
- It permits foreign-key limitations and uniqueness.
- Namespace integration enables distinct sections of a document to adhere to several schemas.

Apart from the functionalities that we have observed, XML Schema facilitates numerous more aspects that DTDs lack, such the following:

- In order to construct specialised types, it permits types to be constrained, for example by defining minimum and maximum values.
- It makes it possible to expand complex types through inheritance.

4.1.5 Querying XML Data

Tools for efficient XML data management are becoming more and more crucial as more applications rely on XML for data exchange, mediation and archiving. Tools for XML data transformation and querying are especially crucial for extracting information from massive XML data sets and converting data across various XML representations (schemas). An XML document may be the result of an XML query, just as a relation is the result of a relational query. Thus, it is possible to integrate transformation and querying into a single tool. We go over the XPath and XQuery languages in this section:

- As a language for path expressions, XPath serves as XQuery's foundation.

Notes

- The common language for XML data querying is called XQuery. Although it is based on SQL, it differs greatly since it must handle nested XML data. XPath expressions are also integrated into XQuery.

Another language meant for XML transformation is called XSLT. But rather than being utilised in data management systems, it is mainly employed in document formatting apps.

Tree Model of XML

All these languages use an XML data tree model. A tree is used to represent an XML document, with nodes standing in for elements and attributes. Subelements or the element's characteristics may be the child nodes of an element node. Accordingly, every node—either an attribute or an element—apart from the root element has an element for a parent node. The order of elements and attributes in the XML document is modelled by the ordering of children of nodes in the tree. The phrases parent, child, ancestor, descendant and siblings are used in the tree model of XML data.

An element's text content can be represented as a text-node child of the element. Text-containing elements with numerous text-node children can be divided up by intervening subelements. An element with the text "This is a <bold> wonderful </bold> book" might, for example, have two text node children that correspond to "this is a" and "book," plus a subelement child that matches the element bold. Since data representation does not frequently employ such structures, we will assume that elements do not contain both text and subelements.

XPath

Path expressions are how XPath uses to address different areas of an XML document. One way to think of the language is as an expansion of object-oriented and object-relational databases' basic path expressions.Our definition is based on XPath 2.0, the most recent version of the XPath standard. Whereas location steps in SQL are separated by the "." operator, a path expression in XPath is a series of location steps separated by "/". A set of nodes is what comes out of a path expression.

```
/university-3/instructor/name
returns these elements:
<name>Srinivasan</name>
<name>Brandt</name>
The expression:
/university-3/instructor/name/text()
```

returns the same names, but without the enclosing tags. Evaluation of path expressions occurs from left to right. Similar to a directory structure, the document's root is indicated by the starting '/'. Keep in mind that the document tag <university-3> is "above" this abstract root. An ordered collection of document nodes makes up the path's result at any given time when a path expression is evaluated. There is only one node in the "current" collection of elements at first—the abstract root. The nodes corresponding to elements of the supplied name that are children of elements in the current element set make up the step result when the next step in a path expression is an element name, such as instructor. The current element set for the subsequent stage of the path expression evaluation is then made up of these nodes. Consequently, the phrase

Notes

/university-3

yields a single node that is associated with the:

<university-3>

tag, while:

/university-3/instructor

gives back the two nodes that match the

instructor

elements that are children of the:

university-3

node.

After the final stage of path expression assessment, the set of nodes is the outcome of a path expression. Each step's returned nodes show up in the same order as they do in the document. The number of nodes in the node set can change with each step because numerous children may share the same name. The "@" symbol can also be used to access attribute values. As an example, the query /university-3/course/@course id yields a collection of all course element course id attribute values. IDREF links are not followed by default; we will address IDREFs in a later section. In addition, XPath has several other features:

- Selection predicates are enclosed in square brackets and can come after any step in a path. As an illustration,

  /university-3/course[credits >= 4]

  returns course materials with a credit value of four or more, but

  /university-3/course[credits >= 4]/@course id

  gives back the course IDs for those courses.

  If we were to delete the single ">= 4" from the example above, the expression would return the course identifiers of all courses that had a credits subelement, regardless of its value. This is an example of how we can verify the existence of a subelement by listing it without performing any comparison operations.

- A number of functions from XPath can be included in predicates. These functions include count(), an aggregate function that counts the number of nodes matched by the expression it is applied to and test(), which determines the current node's position in the sibling order.

  /university-2/instructor[count(./teaches/course)> 2]

  Teachers with more than two courses taught are reimbursed. While the function not(...) can be used for negation, boolean connectives and and or can be utilised in predicates.

- The node (if any) having an attribute of type ID and value "foo" is returned by the function id("foo"). Even collections of references, or even strings with several references separated by blanks, like IDREFS, can be subjected to the function id. For example, the route:

  /university-3/course/id(@dept name)

  gives back every department element that was referred to by the course elements' dept name attribute, however

Notes

/university-3/course/id(@instructors)

provides the instructor aspects that were mentioned in the course elements' instructor attributes.

- It is possible to unione expression results by using the | operator.

  /university-3/course[@dept name="Comp. Sci"] |

  /university-3/course[@dept name="Biology"]

  Nevertheless, it is not possible to nest the | operator inside other operators. It's also important to note that the union's nodes are returned in the document's original order of appearance.

- "//" allows an XPath expression to skip multiple node levels. The formula /university-3// name, for example, returns all name elements anywhere under the /university-3 element, independent of the components they contain and independent of the number of levels of enclosing elements that exist between the university-3 and name elements. This example shows how necessary data can be located without having a complete understanding of the schema.
- A path step need not limit itself to choosing from the offspring of the nodes in the current node set. As it happens, a step on the route might go in a lot of different directions. Examples of these include parents, siblings, ancestors and descendants. While ".." identifies the parent, "//," as previously mentioned, is a concise form for identifying "all descendants." We leave out specifics.
- The built-in function doc(name), where name can be either a file name or a URL, returns the root of a named document. To access the contents of the page, utilise the root that the function returned in a path expression. As a result, rather than applying a path expression to the current default document, it can be applied to a specified document.

Similar to doc, the function collection(name) yields a collection of documents that are identified by name. The function collection can be used, for example, to access an XML database, which can be considered as a collection of documents; the following element in the XPath expression would choose the relevant document(s) from the collection.

The majority of our examples presume that the expressions are evaluated within the framework of a database, which implicitly offers a set of "documents" that are the subject of the evaluation of XPath expressions. We don't need to use the doc and collection functions in these situations.

XQuery

The World Wide Web Consortium (W3C) has developed XQuery as the standard query language for XML. XQuery 1.0, published as a W3C recommendation on January 23, 2007, serves as the foundation for our conversation.

FLWOR Expressions

Although they are modelled after SQL queries, XQuery queries are very different from SQL. The sections are arranged as follows: for, let, where, order by and return. The five portions are represented by the letters in "FLWOR," which stands for "flower" expressions.

Based on the above code's XML document, a straightforward FLWOR expression that yields course identifiers for courses with more than three credits is displayed below. It makes use of ID and IDREFS:

for $x in /university-3/course

Notes

```
let $courseId := $x/@course id
where $x/credits > 3
return <course id> { $courseId }
</course>
```

Similar to the SQL from clause, the for clause defines variables that span the outcomes of XPath expressions. Similar to what the SQL from clause accomplishes, when many variables are supplied, the results include the Cartesian product of all possible values for the variables. For ease of representation, the let clause only permits the assignment of XPath expression results to variable names. Similar to the SQL where clause, the where clause runs extra checks on the joined tuples that come from the for clause. Similar to the SQL order by clause, the order by clause permits output sorting. Lastly, the return clause permits the creation of XML results.

A FLWOR query does not have to include every clause; it might, for instance, only include the for and return clauses and leave out the let, where and order by clauses. There was no order by clause in the previous XQuery query. Actually, because this query is so simple, the let clause can be removed with ease and $x/@course id could take the place of the variable $courseId in the return clause. Note further that, since the for clause employs XPath expressions, choices may occur within the XPath expression. An analogous query might therefore only contain the for and return clauses:

```
for $x in /university-3/course[credits > 3]
return <course id> { $x/@course id } </course id>
```

But the allow clause makes complicated queries simpler. Note also that variables assigned by let clauses may include sequences with multiple elements or values, if the path expression on the right-hand side returns a sequence of multiple elements or values. Note how the return clause uses curly brackets (“\}”). When XQuery detects an element, like the beginning of an expression, it evaluates the text inside curly brackets as an expression and interprets the rest of the text as ordinary XML text. Therefore, the result would contain many copies of the text “$x/@course id,” each surrounded in a course id tag, if the curly brackets in the return clause above were removed. Nonetheless, the contents of the curly brackets are regarded as expressions that need to be assessed. Keep in mind that even if the curly brackets are included in quotations, this convention still holds true. As a result, by changing the return clause to the following, we may change the previous query to return an element with tag course and the course identification as an attribute:

```
return <course course id=“{$x/@course id}” />
```

XQuery offers an additional method of building elements through the use of the element and attribute constructors. The query would return course elements with course id and dept name as attributes and title and credits as subelements, for instance, if the return clause in the previous query was changed to the following one.

```
return element course {
attribute course id {$x/@course id},
attribute dept name {$x/dept name},
element title {$x/title},
element credits {$x/credits}
}
```

Notes

Joins

Similar to SQL, XQuery also specifies joins. This is how XQuery may be used to write the join of the course, instructor and teaches items in the code below:

```
for $c in /university/course,
$i in /university/instructor,
$t in /university/teaches
where $c/course id= $t/course id
and $t/IID = $i/IID
return <course instructor> { $c $i }
</course>
```

The selections listed as XPath selections can be used to express the same query:

```
for $c in /university/course,
$i in /university/instructor,
$t in /university/teaches[ $c/course id= $t/course id
and $t/IID = $i/IID]
return <course instructor> { $c $i } </course instructor>
```

The path expressions in XPath2.0 and XQuery are identical. A single value or element or a list of values or elements can be returned using path expressions. It might not be able to determine if a path expression produces a sequence of values or a single value in the absence of schema information. Such path expressions may engage in comparison procedures such as =, =. An intriguing definition of comparison operations on sequences may be found in XQuery. If the result of $x/credits is a sequence of numbers, for instance, the expression $x/credits > 3 would have the normal interpretation. However, if the result is a single value, the expression evaluates to true if at least one of the values is more than 3. Comparably, if any value returned by the first expression equals any value returned by the second expression, the equation $x/credits = $y/credits evaluates to true. You can use the operators eq, ne, lt, gt, le and ge in place of this behaviour if it is inappropriate. If either of their inputs is a sequence with more than one value, these trigger an error.

4.1.6 Storage of XML data

XML data must be stored for a variety of purposes. Creating a special-purpose database specifically designed to hold XML data is one method of storing it, while another is storing it as documents within a file system. Transforming the XML data into a relational representation and storing it in a relational database is an additional strategy.

Non-relational Data Stores

For the purpose of storing XML data in nonrelational data-storage systems, there are various options available:

- Store in flat files. A flat file is a natural storage technique for XML since it is essentially a file format. Many of the disadvantages of basing database applications on file systems also apply to this technique. Specifically, it is deficient in data security, concurrent access, atomicity and isolation. Nonetheless, accessing and querying XML

Notes

data stored in files is very simple due to the abundance of XML tools that operate on file data. Thus, for certain applications, this storage format might be adequate.

- Create an XML database. Databases that employ XML as their primary data model are known as XML databases. The Document Object Model was implemented in early XML databases using an object-oriented database built on C++. This offers a common XML interface and permits a large portion of the object-oriented database infrastructure to be reused. The addition of XQuery or other XML query languages offers declarative querying. In some implementations, a store manager that supports transactions has served as the foundation for the whole XML storage and querying system.

While there are a number of databases created expressly to hold XML data, creating a fully functional database system from scratch is an extremely difficult undertaking. In addition to XML data storage and querying, this kind of database needs to offer other database functions including security, transaction processing, client-side data access and several management options. Rather, it makes sense to implement XML data storage and querying either on top of or as a layer parallel to the relational abstraction and to use an existing database system to offer these services.

Relational Databases

There are many advantages to storing XML data in relational databases so that it may be accessed from current applications, as relational databases are frequently utilised in applications already in use. If the data were initially generated from a relational schema and XML is only used as a data interchange medium for relational data, then converting XML data to relational form is typically simple. But in many applications, the XML data are not created from a relational schema and it might not be easy to convert the data to a relational form for storing. Specifically, components that are nested and recurrent (equivalent to set-valued attributes) make relational data storage of XML data more difficult. There are a number of different strategies that we go over here.

Store as String

In a relational database, small XML documents can be kept as string (clob) values in tuples. It is possible to manage large XML documents where the top-level element has numerous children by storing each child element as a string in a different tuple. The XML data in the previously stated code, for example, might be stored as a set of tuples in a relation elements(data), where each tuple's attribute data stores one XML element (department, course, teacher, or teaches) in string form.

Although the aforementioned format is user-friendly, the database system is unaware of the items' schema. Consequently, direct data querying is not feasible. In fact, without scanning every tuple in the relation and looking through the text contents, it is not even possible to implement basic selects like discovering every department element or the department element with the department name "Comp. Sci." Keeping various element kinds in separate relations and storing the values of a few key elements as relation attributes to facilitate indexing are two partial solutions to this issue. For example, the department elements, course elements, instructor elements and teaches elements, each with an attribute data, would constitute the relations in our case. In order to record the values of some subelements, like department name, course id, or name, each relation may contain additional attributes. As a result, this representation may effectively respond to a query that needs department elements with a given department name. This method is dependent on type information about XML data, like the data's DTD.

Notes

Function indices are supported by some database systems, including Oracle and can aid in preventing attribute replication between relation and XML string attributes. Function indices can be constructed on the outcome of performing user-defined functions on tuples, in contrast to conventional indices, which are based on attribute values. For example, a user-defined function that returns the value of the department name subelement of the XML string in a tuple can have a function index created on it. After that, the index can be utilised exactly as an index on a department name attribute. The disadvantage of the aforementioned methods is that a significant portion of the XML data is kept in strings.

Tree Representation

Any XML data can be saved using a relation and represented as a tree:

nodes(id, parent id, type, label, value)

Every attribute and element in the XML data has a distinct identification. For every element and attribute, a tuple containing the element's or attribute's identifier (id), parent node's identifier (parent id), type of node (either attribute or element), element or attribute name (label) and element or attribute text value (value) is placed into the nodes relation.

An additional attribute position can be added to the nodesrelation to show the child's relative position among the parent's children if the order of the elements and attributes needs to be maintained. One benefit of this approach is that all XML data can be immediately represented as relational data and many XML queries can be converted into relational queries and run within the database system. The disadvantage of this approach is that each element becomes fragmented, necessitating numerous joins to reassemble subelements into an element.

Map to Relations

Using this method, relations and attributes are mapped to XML elements with established schema. Unknown schema elements are kept as strings or as a tree. For every element type (including subelements) whose schema is known and whose type is complicated (i.e., comprises attributes or subelements), a relation is formed. If the document's root element is attribute-free, it can be disregarded in this phase. The following is a definition of the relation's attributes:

- These elements' characteristics are all saved as string-valued relational attributes.
- An attribute is added to the relation to indicate the subelement if it is a simple type subelement (i.e., cannot have attributes or subelements). If the subelement contained an XML Schema type, an equivalent SQL type may be used instead of the relation attribute's default string value. For instance, the subelements department name, building and budget of the element department become attributes of a relation department when applied to the element department in the schema of the data in the code given above.
- If not, the same rules are applied recursively to its subelements to produce a relation that corresponds to the subelement. Additionally:
  - ❖ The relations that represent the element get an additional identifier attribute. (Even if an element contains multiple subelements, the identification property is added only once.)
  - ❖ The identifier of the parent element is stored in an attribute called parent id, which is added to the relation that represents the subelement.
  - ❖ An attribute position is added to the relation representing the subelement if ordering is to be maintained.

Notes

Alternatives to this strategy are conceivable. By transferring all of their characteristics into the parent relation, relations corresponding to subelements that can occur only once can be "flattened" into the parent relation. There are references to various methods for representing XML data as relations in the bibliographical notes.

Publishing and Shredding XML Data

The data that is exchanged between business applications via XML typically comes from relational databases. To be exported to other apps, data in relational databases needs to be published, or transformed into XML format. New data must be shredded, or transformed back from XML to a relational database in a normalised format. Although publishing and shredding activities can be carried out by application code, since these actions are so often, it is preferable if possible for the conversions to be carried out automatically without the need for application code. Database providers have put a lot of work into making their solutions XML-capable.

An automatic technique for publishing relational data as XML is supported by an XML-enabled database. Data publication may involve simple or complicated mapping. Every table row might have an XML element created for it, with each column in that row becoming a subelement of the XML part, thanks to a straightforward relation to XML mapping. With the help of such a mapping, the XML structure in the code above can be produced from a relational representation of university data. It is simple to automatically generate such a mapping. One way to handle an XML representation of relational data is to treat it like a virtual XML document, which can then be queried using XML queries. A more intricate mapping would make the creation of nested structures possible. To make it simple to create nested XML output, extensions for SQL that include nested queries in the select clause have been developed.

In order to transform XML data into a relational representation, mappings must also be specified. The mapping needed to shred XML data derived from a relational representation is simply the opposite of the mapping used to publish the data.

Native Storage within a Relational Database

Some relational databases support native storing of XML. These systems don't transform XML data into relational form; instead, they save it as strings or in more effective binary forms. Although the fundamental storage technique may be provided by the CLOB and BLOB data types, a new data type called xml is developed to represent XML data. XML data can be queried using XML query languages like XPath and XQuery. A relation with an attribute of type xml can be used to store a collection of XML documents; each document is saved as a value of type xml in a distinct tuple. To index the XML data, special-purpose indices are constructed. For XML data, a number of database systems offer native support. They enable the embedding of XQuery queries within SQL queries and offer an XML data type. A single XML document can be the target of an XQuery query, which can also be incorporated in a SQL query to enable it to run on all the documents in a collection, each of which is kept in a different tuple.

SQL/XML

Relational databases continue to be the primary storage for structured data, notwithstanding the widespread usage of XML for data transmission. Converting relational data to an XML representation is frequently necessary. This necessity led to the development of the SQL/XML standard, which specifies a common extension of SQL that enables the production of nested XML output. The standard consists of multiple components, such as SQL query language extensions and a standard method for mapping

Notes

relational schemas to XML schemas and SQL types to XML Schema types. An XML schema with the outermost element department, each tuple mapped to an XML element row and each relation attribute mapped to an XML element of the same name would be the example of the SQL/XML representation of the department relation (with some conventions to resolve incompatibilities with special characters in names). In a similar way, a whole SQL schema with several relations can likewise be mapped to XML. The SQL/XML representation of a portion of the university's data, including the department and course relationships, is displayed in the code below. Several operators and aggregate operations are added to SQL by SQL/XML, enabling the creation of XML output straight from the expanded SQL.

```
<university>
    <department>
     <row>
     <dept name> Comp. Sci.
     </dept>
     <building> Taylor </building>
     <budget> 100000 </budget>
    </row>
    <row>
     <dept name> Biology
    </dept>
    <building> Watson </building>
    <budget> 90000 </budget>
</row>
</department>
<course>
    <row>
     <course id> CS-101
    </course>
    <title> Intro. to Computer Science </title>
    <dept name> Comp. Sci
</dept>
<credits> 4 </credits>
</row>
<row>
    <course id> BIO-301
</course>
<title> Genetics </title>
<dept name> Biology
</dept>
<credits> 4 </credits>
</row>
<course>
</university>
```

Notes

The following query demonstrates how to build XML elements using the xmlelement function and attributes using xmlattributes.

```
select xmlelement (name “course”,
xmlattributes (course id as course id, dept name as dept name),
xmlelement (name “title”, title),
xmlelement (name “credits”, credits))
from course
```

With the course identification and department name as properties and the title and credits as subelements, the aforementioned query generates an XML element for every course. Missing the teacher characteristic. The xmlattributes operator constructs the XML attribute name using the SQL attribute name, which can be altered using an as clause as illustrated.

XML structure creation is made easier with the help of the xmlforest operator. With the exception of creating a forest (collection) of subelements rather than a list of attributes, its syntax and behaviour are similar to that of xmlattributes. It accepts several inputs and creates an element with the XML element name equal to the SQL name of the attribute. Subexpression-created components can be concatenated into a forest using the xmlconcat operator. An attribute is ignored when the SQL value used to create it is null. When building an element’s body, null values are left out. Additionally, SQL/XML offers the aggregate function xmlagg, which takes a collection of data as input and outputs an XML element forest (collection). With the help of the following query, an element with all of the department’s courses as subelements is created for each department that has a course. The aggregate function is applied to every course in every department because the query has a clause group by department name. This results in a series of course id items.

```
select xmlelement (name “department”,
dept name,
xmlagg (xmlforest(course id)
order by course id))
from course
group by dept name
```

The previous query demonstrates how to organise the sequence produced by xmlagg using SQL/XML. For further information on SQL/XML, see the references in the bibliographical notes.

4.1.7 XML Applications

We now describe a number of uses for XML that involve data communication and storage, as well as Web service (information resource) access.

Storing Data with Complex Structure

Structured data that is difficult to model as relations is needed for many applications. Take user preferences, for instance, which a program like a browser has to store. The home page, security settings, language settings and display settings are just a few of the many sections that often need to be entered. Certain fields, such a list of reliable websites or maybe ordered lists like bookmarks, are multivalued. Applications have historically stored such data using a textual representation of some kind. Most of these programs these days

Notes

prefer to save this kind of setup data in XML format. It takes work to build and develop parsers that can read the file and transform the contents into a format that a program can use in place of the ad hoc textual representations that were previously used. Both of these processes are omitted in the XML representation.

These days, storing documents, spreadsheet data and other data included in office application packages is commonly done using XML-based representations. Two document representation formats based on XML are the Office Open XML (OOXML) format, supported by the Microsoft Office suite and the Open Document Format (ODF), supported by the Open Office software package and other office suites. These are the two formats that are most frequently used to represent editable documents. Data with complicated structures that need to be shared between various application components are also represented using XML. A database system might use XML, for instance, to represent a query execution plan, which is a relational algebra expression plus additional information on how to carry out operations. This eliminates the need for a common data structure by enabling one component of the system to develop the query execution plan and another to display it. As an illustration, the data might be created on a server system and transferred to a client system for display.

Standardised Data Exchange Formats

For a range of specialised applications, including business applications like banking and shipping as well as scientific applications like chemistry and molecular biology, XML-based standards for data representation have been developed. A few instances are:

- The chemical industry needs knowledge about chemicals, such as their molecular structure and a number of critical qualities, such as boiling and melting points, calorific values and solubility in various solvents. A standard for expressing this kind of data is ChemML.
- When shipping, carriers of products, as well as customs and tax officials, require shipment documents that include comprehensive details on the commodities being transported, including who is carrying them, where they are being shipped and how much they will cost.
- An online marketplace in which business can buy and sell items [a so-called business-to-business (B2B) market] requires information such as product catalogues, containing complete product descriptions and price information, product inventories, quotes for a proposed sale and purchase orders. For instance, XML schemas and semantics for data representation as well as message exchange standards are defined by the RosettaNet standards for e-business applications.

Such complicated data requirements would produce a huge number of relations that do not immediately correlate to the objects that are being modelled if normalised relational schemas were used to model them. Since there are frequently a lot of attributes in relations, it is helpful to explicitly represent attribute/element names and values in XML to prevent attribute misunderstanding. At the potential cost of duplication, nested element representations assist in reducing the number of relations that must be represented and the number of joins necessary to obtain the necessary information. For example, listing departments with course items nested within department elements in our university example.

Web Services

Applications frequently need information from databases that are used by multiple departments within the same organisation or from outside the organisation. In many of these cases, the external department or organisation is prepared to supply restricted

Notes

information through pre-established interfaces but is unwilling to grant direct SQL access to its database. Organisations offer Web-based forms where users can input values and get needed information in HTML form when the information is intended for direct human use. However, in many cases, software programs rather than end users need to access this kind of information. It's obvious that XML results for a query must be provided. Furthermore, it seems sense to provide the query's input values in XML format as well.

Essentially, the information provider specifies processes whose input and output are both in XML format. Since the HTTP protocol is widely used and can get through firewalls employed by institutions to filter out undesired traffic from the Internet, it is utilised to convey input and output data.

A standard for calling procedures is defined by the Simple Object Access Protocol (SOAP), which uses XML to represent the input and output of the procedure. For the purpose of displaying the procedure name and result status indicators, such as failure/error indicators, SOAP defines a standard XML structure. SOAP XML headers contain application-specific XML data that contains the operation parameters and results.

For SOAP, the transport protocol of choice is usually HTTP; however, message-based protocols, like email via the SMTP protocol, can also be employed. These days, many people use the SOAP standard. For instance, SOAP-based processes are offered by Google and Amazon to perform searches and other tasks. Other programs that offer users higher-level services may call upon these operations. Because the SOAP standard is not dependent on the underlying programming language, a website using C# can call a Java-based service. Similarly, a site using C# can call a service that uses another language.

A Web service is a website that offers this set of SOAP processes. A number of standards have been established to facilitate Web services. The language used to describe a Web service's capabilities is called the Web Services Description Language (WSDL). By defining the available functions and their input and output types, WSDL offers features that traditional programming languages' interface definitions, also known as function definitions, do not. Furthermore, WSDL enables the specification of the network port number and URL to be utilised when launching the Web service. A standard known as Universal Description, Discovery and Integration (UDDI) also outlines how a program can search a directory of Web services that meet its criteria and how to establish a directory of available Web services.

The benefit of Web services is demonstrated in the following example. When an airline defines a Web service, it means that it offers a collection of methods that a travel website can use to obtain flight schedules, price and booking information. The travel website may communicate with many Web services offered by various hotels, airlines and other businesses in order to give customers travel information and facilitate bookings. Through providing support for Web services, the separate businesses enable the construction of a valuable service on top, thereby integrating the individual services. Instead of contacting several different websites, users can book their trip arrangements by interacting with a single website.

In order to use a Web service, a client must first create and deliver the service the proper SOAP XML message. Once the client receives the XML-encoded result, it must extract data from the result. Standard APIs are available for creating and extracting data from SOAP messages in languages like Java and C#.

Data Mediation

One type of mediation application is comparison shopping, which involves gathering information about products, pricing, shipping and inventory from multiple websites that

Notes

sell a specific item. The combined information that is produced is far more valuable than the individual information that is provided by a single website. In the banking arena, an analogous application is a personal financial manager. Think about a customer that has numerous accounts to handle, including retirement, credit card and bank accounts. Assume that these accounts might be kept at various establishments.

It is quite difficult to provide centralised control for every customer's account. In order to solve the issue, XML-based mediation retrieves an XML representation of the account data from the websites of the financial institutions where the person has accounts. If the organisation exports this data in a common XML format—for instance, as a Web service—it could be readily retrieved. For those who don't, XML data is created from HTML Web pages that the website returns using wrapper software. Wrapper programs need continual maintenance, since they depend on formatting features of Web pages, which change constantly. Nevertheless, the benefit given by mediation frequently compensates the effort necessary to design and maintain wrappers.

A mediator application is used to bring all of the collected data under a single schema after the fundamental tools for extracting data from each source are available. Given that multiple websites may organise the same information differently, this might necessitate extra transformation of the XML data from each site. Additionally, they could refer to the same information under different names (such as account ID and account number) or they might refer to the same information under multiple identities. In addition to providing code to convert data between several forms, the mediator must select a single schema that encompasses all necessary information.

4.1.8 Semi Structured Data Model

Semistructured data models allow data to be specified with distinct sets of attributes for individual data items of the same type. This is not the case with the previously discussed data models, where each data item of a specific kind needs to contain the same set of properties.

Originally intended to provide markup information to text documents, the XML language has gained prominence due to its usage in data interchange. Data with nested structures can be represented using XML, which also offers a great deal of flexibility in data structuring—a feature that is crucial for some non-traditional data types.

Think of a collection of Web documents that have links to other documents within them. These documents are not entirely unstructured, but their hyperlink pattern is irregular between documents, so they cannot be readily represented in the relational data paradigm. Although a bibliography file is primarily composed of unstructured text, it does have some organisation thanks to fields like author and title. Many types of data are neither entirely unstructured nor entirely structured, however other data—such as video and audio streams and image data—is entirely unstructured. Semistructured data is the term used to describe data that has some organisation. A significant and expanding source of semistructured data are XML documents and the theory of semistructured data models and queries may provide the framework for XML.

There are numerous causes for semistructured data. First, the data structure may be implicit, concealed, unknown, or ignored by the user. Second, think about the challenge of integrating data from several heterogeneous sources, where significant issues with data transformation and interchange arise. To combine data from many data sources, such as flat files and legacy systems, we require a highly flexible data model; structured data models are sometimes too inflexible. Third, although we can never fully query a structured database

Notes

without understanding its schema, there are situations in which we wish to do so. For instance, in a relational database system, we are unable to formulate the question, “Where in the database can we find the string Malgudi?” unless we are aware of the structure.

For semistructured data, every data model that has been suggested views the data as a labelled graph of some sort. In the graph, characteristics are represented by edges, while complex objects or atomic values are represented by nodes. The information in the graph self-describes; neither a distinct schema nor an additional explanation are needed. Take into consideration, for instance, the graph in Figure below, which depicts a portion of the XML data from the code below. The outermost element, BOOKLIST, is represented by the graph’s root node. The node has three outgoing edges that are labelled with the element name BOOK, since the list of books comprises of three unique books.

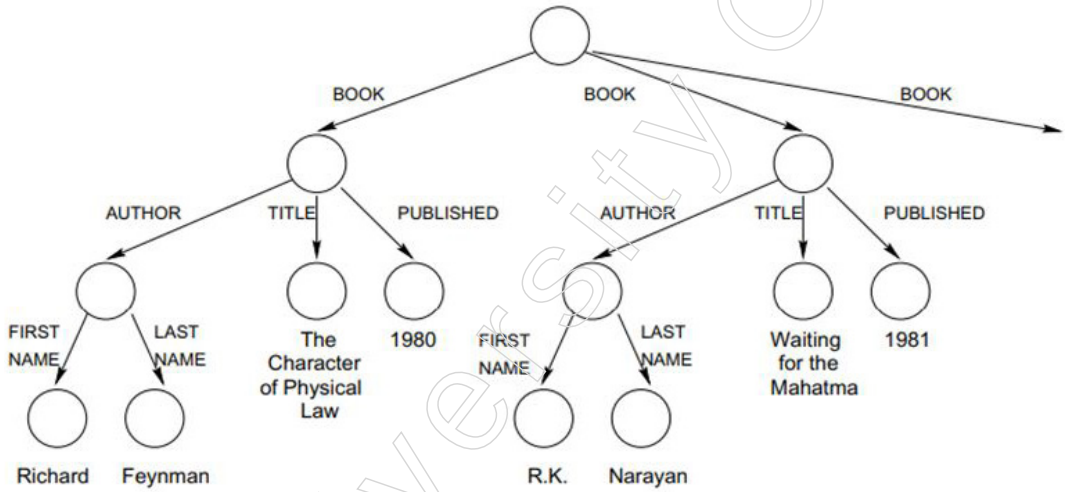


Figure: The Semistructured Data Model

Image Source: database-management-systems-raghu-ramakrishnan

```
<?XML version=”1.0” encoding=”UTF-8” standalone=”yes”?>
<!DOCTYPE BOOKLIST SYSTEM “books.dtd”>
<BOOKLIST>
    <BOOK genre=”Science” format=”Hardcover”>
     <AUTHOR>
     <FIRSTNAME>Richard</FIRSTNAME>
     <LASTNAME>Feynman</LASTNAME>
     </AUTHOR>
     <TITLE>The Character of Physical Law</TITLE>
     <PUBLISHED>1980</PUBLISHED>
    </BOOK>
    <BOOK genre=”Fiction”>
     <AUTHOR>
     <FIRSTNAME>R.K.</FIRSTNAME>
     <LASTNAME>Narayan</LASTNAME>
     </AUTHOR>
     <TITLE>Waiting for the Mahatma</TITLE>
```

Notes

```
<PUBLISHED>1981</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
<AUTHOR>
<FIRSTNAME>R.K.</FIRSTNAME>
<LASTNAME>Narayan</LASTNAME>
</AUTHOR>
<TITLE>The English Teacher</TITLE>
<PUBLISHED>1980</PUBLISHED>
</BOOK>
</BOOKLIST>
```

This brings us to our discussion of the object exchange model (OEM), one of the suggested data models for semistructured data. A triple, comprising an object's value, type and label, is used to describe each item. Since each object has a label that can be thought of as the column name in the relational model and each object has a type that can be thought of as the column type in the relational model, the object exchange model is effectively self-describing. In order to fulfil two functions, labels in the object exchange model should be as informative as possible: Both the meaning of an object and its identification can be communicated through them. The last name of an author, for instance, can be represented as follows:

<lastName, string, "Feynman">

Smaller things are broken down hierarchically from more complex ones. An author's name, for instance, may have both a first and a last name. The following is a description of this item:

```
<hauthorName1, set, {f irstname1, lastname}>
firstname is <firstName, string,"Richard">
lastname1 is <lastName, string,"Feynman">
<bookList, set, {book1, book2, book3}>
book1 is <book, set, {author1, title1, published1}>
book2 is <book, set, {author2, title2, published2}>
book3 is <book, set, {author3, title3, published3}>
author3 is <author, set, {f irstname3, lastname3}>
title3 is <title, string, "The English Teacher">
published3 is <published, integer, 1980>
```

4.1.9 Implementation Issues

Recent years have seen a significant amount of research focused on database systems supporting semistructured data and given the commercial success of XML, this focus is expected to continue. Since most conventional storing, indexing and query processing algorithms presume that the data adheres to a regular schema, semistructured data presents unique issues. Should semistructured data, for instance, be translated into the relational model and then stored in a relational database system? Or does higher

Notes

performance come from a storage subsystem designed specifically for semistructured data? How can semistructured data be indexed? Which query processing techniques work best with a query language such as XML-QL? These questions are being researched in an effort to provide answers.

4.1.10 Indexes for Text Data

We refer to a database that contains documents as a collection when it is assumed to be a text database. To keep things simple, we'll suppose that there is only one relation in the database and one field of type document in the relation structure. Consequently, there is exactly one document in every record in the relation. In actuality, the relation schema would have more data like the document's creation date, a potential classification, or a field containing keywords that describe the content. Legal documents, media articles and other kinds of documents are kept in text databases.

Asking for all papers that include a specific term is made possible by a significant class of keyword search queries. These days, the most popular type of inquiry on the Internet is this one, which is handled by several search engines like AltaVista and Lycos. A query asking for documents containing "car" will also return documents containing "automobile," as some systems keep a list of synonyms for key phrases and return documents that contain the requested keyword or one of its synonyms. "Find all documents that have keyword1 AND keyword2" is a more sophisticated search query. By utilising AND, OR and NOT to create composite searches, we are able to rank the retrieved documents based on how close the query terms are to the document.

Boolean and ranked inquiries are the two most used kinds of queries for text databases. The following boolean expression, known as conjunctive normal form, is provided by the user in a boolean query:

$(t11 \vee t12 \vee ... \vee t1i1) \vee ... \vee (tj1 \vee t12 \vee ... \vee t1ij)$,

where the tij are individual query terms or keywords. There are j conjuncts in the query and each one is made up of many disjuncts. The expression $(t11 \vee t12 \vee ... \vee t1i1)$ is the first conjunct in our query; it is made up of i1 disjuncts. There is a natural interpretation for queries in conjunctive normal form. Documents involving many concepts are the query's output. Every conjunct represents a single notion, while the several words that make up a conjunct represent various phrases for the same concept.

The structure of ranked queries is relatively similar. A list of documents ranked according to how relevant they are to the user's list of terms is what emerges from a ranked query, in which the user additionally provides a list of words. It is a challenging task to determine when and how relevant a content is to a set of user keywords. Database administration is strongly related to the topic of information retrieval, which includes algorithms for evaluating queries of this type. The objective of information retrieval systems, similar to database systems, is to allow users to query vast amounts of data; however, the emphasis has been on unstructured document collections. Since most applications' data is static, information retrieval systems have historically ignored updates, concurrency control and recovery.

Recall, or the proportion of pertinent documents in the database that are returned in response to a query and precision, or the percentage of retrieved documents that are relevant to the query, are the two metrics used to assess these information retrieval systems.

Since users may now access millions of documents, information retrieval has gained new momentum with the introduction of the Web. Finding the information they need is a

Notes

basic function and without effective search tools, users would become overwhelmed. A Web search engine generates a centralised index for documents that are maintained at several sites. An information retrieval system's index basically consists of hkeyword,documentidi pairs, sometimes with additional fields like the number of times a keyword appears in a document.

This section's remaining content focuses on boolean queries. We present two index designs that facilitate the effective evaluation of boolean queries. The inverted file index is a popular choice because of its efficiency and ease of usage. Its primary drawback is the large space overhead it imposes; the size can increase to 300 times the original file size. The signature file index provides a fast filter that removes the majority of nonqualifying documents with a minimal overhead. However, because the index needs to be sequentially scanned, it scales less well to greater database volumes.

We presume that when the index was created, somewhat dissimilar terms with the same root were stemmed, or examined for the same root. For instance, we take it for granted that documents containing the terms "indexes" and "indexing" will likewise appear in the output of a query on "index." We won't get into the specifics of stemming because it depends on the application.

As an example, let's say we have the four documents that are depicted in the figure below. To keep things simple, we'll assume that the four documents' record identifiers are one through four. Record identifiers are often entries in an address table rather than physical addresses on the disc. An array known as an address table is used to translate logical record identifiers—which are displayed in the figure below—to physical record addresses on disc.

Rid	Document	Signature
1	agent James Bond	1100
2	agent mobile computer	1101
3	James Madison movie	1011
4	James Bond movie	1110

Word	Inverted list	Hash
agent	⟨1, 2⟩	1000
Bond	⟨1, 4⟩	0100
computer	⟨2⟩	0100
James	⟨1, 3, 4⟩	1000
Madison	⟨3⟩	0001
mobile	⟨2⟩	0001
movie	⟨3, 4⟩	0010

Figure: A Text Database with Four Records and Indexes

Image Source: database-management-systems-raghu-ramakrishnan

Inverted Files

An index structure called an inverted file makes it possible to quickly get every document that contains a query phrase. The index keeps track of document identifiers that contain the indexed phrase for each word in an ordered list known as the "inverted list." Take the text database depicted in the preceding Figure, for instance. The record identifiers <1, 3 and 4> are inverted for the query phrase "James," but <3, 4> are for the query term "movie." All query phrases are displayed in reverse order in the above figure.

To expedite the process of locating the inverted list for a given query word, every potential query term is arranged within a secondary index structure, like a hash index or a B+ tree. For the purpose of clarity, we will refer to the second index—the vocabulary index—that makes it possible to quickly retrieve the inverted list for a query phrase. Every potential query term and a pointer to its inverted list are contained in the vocabulary index.

Notes

To assess a query with a single term, one must first navigate through the vocabulary index to get the leaf node entry carrying the term's inverted list address. Subsequently, the list is reversed, the rids are translated into actual document addresses and the associated documents are obtained. The process of evaluating a query that involves many phrases is to retrieve the inverted lists of each query term individually and cross them. The inverted lists should be retrieved in increasing length order to reduce memory use. All pertinent inverted lists are combined to analyse a query including a disjunction of several terms.

Take another look at the sample text database displayed in the above figure. In order to assess the query "James," we first obtain document one, then we query the vocabulary index to locate the inverted list for "James." We obtain the inverted list for the term "Bond" and cross it with the inverted list for the term "James" in order to assess the query "James" AND "Bond." (The terms "Bond" and "James" have two and three lengths, respectively, in their inverted lists.) The first and fourth documents are taken from the list <1, 4>, which is the result of the intersection of the lists <1, 3, 4> and <1, 4>. In order to assess the query "James" OR "Bond," we obtain the two flipped lists in any sequence, then combine the outcomes.

Signature Files

A signature file is another index format for text database systems that provides efficient assessment of boolean queries. Every document in the database has an index record in a signature file. The signature on the document is this index record. The signature width is denoted by b, which is the fixed size of b bits for each signature. How do we choose which sections of a document to include? The words that occur in the document determine which bits are set. By applying a hash function to every word in the document and setting the bits that show up in the hash function's output, we are able to map words to bits. Because the hash function translates both words to the same bit, it should be noted that unless we have a bit for every conceivable word in the vocabulary, the same bit could be set twice by different words. If every bit that is set in one signature S2 is also set in another signature S1, then we say that the two signatures match. Signature S1 has at least the same number of bits set as signature S2, if they match.

We first create the query signature for a query that is a conjunction of terms by running the hash function over each word in the query. The next step is to search the signature file for all documents whose signatures match the query signature. This is done because each document that has a matching signature could be a possible query result. We must extract each possible match and confirm that the document truly contains the query terms because the signature cannot uniquely identify the words that a document contains. A false positive is a document whose signature matches the query signature but does not contain every term in the query. Since the document must be retrieved from disc, parsed, stemmed and examined to see if it includes the query terms, a false positive is an expensive error.

We produce a list of query signatures, one for each phrase in a query that is a disjunction of terms. In order to evaluate the query, the signature file is scanned in order to locate documents whose signatures match any signature in the query signatures list. Keep in mind that there are exactly as many records in the signature file as there are documents in the database, so we must scan the entire file for each query. We can vertically divide a signature file into a series of bit slices to minimise the amount of data that needs to be obtained for each query; we refer to such an index as a bit-sliced signature file. Although we only need to obtain q bit slices for a query with q bits set in the query signature, the length of each bit slice is still equal to the total number of documents in the database.

Notes

Take, for instance, the text database with a signature file of width 4 that is depicted in the above Figure. The picture displays the bits set by all query phrases' hashed values. We first calculate the term's hash value, which is 1000, in order to analyse the query "James." Next, we locate matching index records by scanning the signature file. The first bit is set in all record signatures, as the above figure illustrates. All documents are retrieved and screened for false positives; the document with rid 2 is the sole false positive for this query. (Unfortunately, the signature's initial bit is likewise set by the hashed value of the word "agent.") Think about the questions "James" AND "Bond." Three document signatures and the query signature, which is 1100, are in agreement. We recover one false positive once again. An additional illustration of a conjunctive query is the one that looks at "movie" AND "Madison." There is only one document signature that matches the query signature, which is 0011. There are no retrieved false positives. It is suggested that the reader create a bit-sliced signature file and use the bit slices to assess the sample queries in this paragraph.

Summary

- Web interfaces are crucial components of web development, serving as the user's gateway to interact with web applications. These interfaces are designed to be intuitive, visually appealing and responsive. web interfaces play a pivotal role in shaping the user experience of web applications. Developers need to blend technical proficiency with design principles to create interfaces that are not only functional but also visually appealing and user-centric.
- XML, or Extensible Markup Language, is a versatile and widely used markup language designed to structure, store and transport data in a format that is both human-readable and machine-readable. It provides a set of rules for encoding documents in a format that is both easy to read by humans and straightforward to process by machines.
- XML (Extensible Markup Language) data follows a hierarchical structure, organised in a tree-like format. This structure consists of elements, attributes and text content. Understanding the key components of the XML structure is essential for creating and interpreting XML documents.
- A document schema in XML provides a blueprint or a set of rules that define the structure and constraints of an XML document. It serves as a formal specification, ensuring that XML documents adhere to a predefined structure, data types and relationships. Two common technologies used for defining XML document schemas are Document Type Definition (DTD) and XML Schema Definition (XSD).
- The semi-structured data model is a flexible and versatile approach to data representation that falls between the rigid structure of relational databases and the lack of structure in unstructured data. It allows data to be organised in a hierarchical manner, without requiring a strict schema definition.
- Querying XML data involves extracting specific information from XML documents based on certain criteria. Various query languages and approaches are used to navigate and retrieve data from hierarchical XML structures.

Glossary

- HTML: Hyper Text Markup Language
- CSS: Cascading Style Sheets
- XSLT: Extensible Stylesheet Language for Transformations,
- XSL: Extensible Stylesheet Language
- UX: User Experience

Notes

- UI: User Interface
- SSL: Secure socket layers
- W3C: World Wide Web Consortium
- XSD: XML Schema Definitions
- DTDs: Document Type Definitions
- SOAP: Simple Object Access Protocol
- ODF: Open Document Format
- WSDL: Web Services Description Language
- I/O: Input/Output
- CPU: Central Processing Unit
- IoT: Internet of Things
- SQL: Structured Query Language
- DDBS: Distributed Database System
- DDL: Data Definition Language
- 2PC: Two-Phase Commit
- 3PC: Three-Phase Commit

Check Your Understanding

1. What technology is commonly used for creating responsive and dynamic web interfaces?
   a) HTML
   b) CSS
   c) JavaScript
   d) XML
2. Which of the following is a primary purpose of AJAX (Asynchronous JavaScript and XML) in web interfaces?
   a) Creating static web pages
   b) Enabling asynchronous data exchange with the server
   c) Defining document structure
   d) Styling web pages
3. What role does CSS (Cascading Style Sheets) play in web interfaces?
   a) Defining document structure
   b) Styling and formatting the presentation of HTML elements
   c) Enabling server-side processing
   d) Managing database interactions
4. Which of the following technologies is essential for creating a responsive, mobile-friendly web interface?
   a) PHP
   b) React
   c) Bootstrap
   d) SQL

5. In XML, what does the root element represent?
   a) The first child element
   b) The last child element
   c) The topmost element in the hierarchy
   d) An attribute of an element

Notes

Exercise

1. What do you mean by web interfaces?
2. Define XML and its structure.
3. What do you mean by document schema?
4. Define querying XML data.
5. How to store XML data?

Learning Activities

1. Discuss scenarios where XML is commonly used and highlight its advantages in representing hierarchical data structures.
2. Explain the significance of a document schema in XML. Discuss how a schema defines the structure, constraints and relationships within an XML document. Provide insights into the benefits of using schemas in XML data modeling.

Check Your Understanding- Answers

1. c 2. b 3. b 4. c
5. c

Notes

Module -V: Advance Transactions and Emerging Trends

Learning Objectives

At the end of this module, you will be able to:

- Understand database management system
- Define multilevel transactions and long-lived transactions (saga)
- Define data warehousing and mining
- Define functional dependencies and normal forms
- Analyse different types of database

Introduction

Although the phrase “we are living in the information age” is common, we are actually living in the data age. Every day, terabytes or petabytes of data from business, society, science and engineering, medicine and nearly every other element of daily life flood our computer networks, the World Wide Web (WWW) and numerous data storage devices. The rapid development of effective data gathering and storage methods, as well as the computerisation of our society, are to blame for this tremendous expansion in the volume of data that is currently available.

Globally, businesses produce enormous amounts of data, such as stock trading records, product descriptions, sales promotions, corporate profiles and consumer feedback. For instance, huge retailers with thousands of locations around the world, like Wal-Mart, conduct hundreds of millions of transactions per week. High orders of petabytes of data are continuously produced by scientific and engineering procedures, including remote sensing, process measurement, scientific experiments, system performance, engineering observations and environment surveillance.

Data mining converts a sizable data collection into knowledge. Every day, hundreds of millions of searches are made using search engines like Google. Every query can be seen as a transaction in which the user expresses a demand for information. What innovative and helpful information can a search engine get from such a vast database of user queries gathered over time? It’s interesting how some user search query patterns can reveal priceless information that cannot be learned from examining individual data items alone.

Google’s Flu Trends, for instance, uses particular search phrases as a gauge of flu activity. It discovered a strong correlation between the number of persons who actually have flu symptoms and those who look for information relevant to the illness. When all of the flu-related search searches are combined, a pattern becomes apparent. Flu Trends can predict flu activity up to two weeks sooner than conventional methods by using aggregated Google search data. This illustration demonstrates how data mining may transform a sizable data set into knowledge that can assist in resolving a current global concern.

The market has been overrun with different products by hundreds of suppliers during the last five years. Data modelling, data gathering, data quality, data analysis, metadata and other aspects of data warehousing are all covered by vendor solutions and products. No fewer than 105 top items are highlighted in the Data Warehousing Institute’s buyer’s guide. The market is already enormous and is still expanding.

Almost sure, you’ve heard of data mining. Most of you are aware that technology plays a role in knowledge discovery. It’s possible that some of you are aware of the use of data mining in fields like marketing, sales, credit analysis and fraud detection. You’re all aware,

Notes

at least in part, that data mining and data warehousing are related. Almost all corporate sectors, including sales and marketing, new product development, inventory management and human resources, use data mining.

The definition of data mining has possibly as many iterations as there has supporters and suppliers. Some experts include a wide variety of tools and procedures in the definition, ranging from straightforward query protocols to statistical analysis. Others limit the definition to methods of knowledge discovery. Although not necessary, a functional data warehouse will offer the data mining process a useful boost

5.1 Database Management System

The technology for efficiently storing and retrieving customer data while also including the necessary security elements is known as a database control system, or DBMS. The fundamentals of DBMS, including its architecture, information models, fact schemas, records independence, e-r version, relation version, relational database design, storage and document structure and many other topics, are explained in this tutorial.

Why look at DBMS? Traditionally, file codecs have been used to arrange data. At the time, DBMS was a brand-new concept and extensive research was done to enable it to overcome the limitations of the conventional methods of records control. A modern DBMS contains the following characteristics:

- Actual-world entity − Modern DBMSs are more rational and base their structural architecture on real-global entities. It also uses the characteristics and behaviour. For illustration, a database for a college might also use students as an entity and their age as an attribute.
- Relation-based totally tables − Entities and family members can be added to DBMS to create tables. By simply looking at the table names, a user can determine the database's structure.
- Isolation of facts and application − A database system is really unique compared to its facts. When compared to facts, which the database organises and works on, a database is an active entity. To facilitate its own system, DBMS also stores metadata, or records about information.
- Less redundancy − DBMS adheres to the normalisation rules, which divide a relation when any of its attributes have redundant values. The mathematically sophisticated and medical approach of normalisation removes information redundancy.
- Consistency − Every relation in a database must remain constant in order for there to be consistency. There are methods and tactics that could be used to attempt to leave a database in an inconsistent state. Compared to older uses for storing information, including document-processing systems, a DBMS can provide more consistency.
- Question language − The query language that is available in DBMS makes it easier to retrieve and manage statistics. To retrieve a group of records, one can use as many and as unique filtering options as necessary. In the past, places where record-processing machines were employed were not possible.
- Multiuser and concurrent get entry to − DBMS supports multi-user environments and permits simultaneous access to and manipulation of data. Despite the fact that there exist restrictions on transactions while users are attempting to manage the same statistics object, users consistently ignore them.
- A couple of views − For particular users, DBMS provides a few perspectives. Someone working in the income department may have a unique perspective on the database

Notes

compared to someone in the production department. This option enables the users to view the database with attention in accordance with their needs.

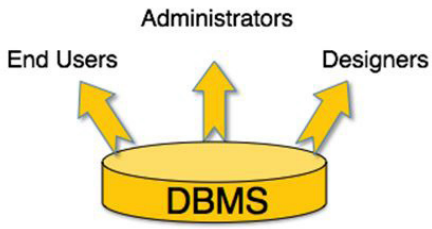
- Security − Customers are prevented from gaining access to statistics of various clients and departments thanks to security features like numerous viewpoints. While entering data into the database and retrieving the same later, DBMS provides methods for imposing limitations.

DBMS provides a wide variety of unique protection features, allowing numerous users to have amasing perspectives with particular characteristics. For example, a person working in the income department is unable to access the data belonging to the purchase branch. The amount of information that the sales department must display to the customer may also be controlled. Because a DBMS is not kept on the disc like traditional report structures, it is exceedingly difficult for criminals to interfere with the code.

Users

Users with unique privileges and permissions use the average DBMS for unique purposes. A few clients have retrieved records and a few have given them back. The following categories may be used to broadly group DBMS users:

- Administrators − Directors are responsible for maintaining the DBMS and running the database. They are responsible for monitoring how and by whom it should be used. They design user entry profiles and use barriers to maintain isolation and enforce security. Directors also take care of DBMS resources including system licences, necessary equipment and upgrades to other software and hardware.
- Designers − The group of people known as designers works solely on designing a portion of the database. They keep a close eye on what data has to be preserved and in what format. They are able to recognise and organise the full group of entities, relatives, limitations and views.
- Give up customers − Give-up users are those who unquestionably benefit from purchasing a DBMS. Leave clients can range from basic viewers who are familiar with the logs or market quotes to cutting-edge users like business analysts.



DBMS Benefits

The following are benefits of a DBMS system:

1. Information redundancy

In contrast to standard report-system garages, DBMS statistics redundancy may be very minimal or nonexistent. Redundant records are created as a result of storing the same data unnecessarily in unusual places. Because all data in DBMS is saved in one location rather than being created by individual users and for each software programme, data redundancy is reduced or eliminated. For example, utility A and alertness B share the same user profile and we need to store personal information about the user, such as call, age, address, start date and so on. Not to mention, this person has access to different software, so in a traditional file-based system, there would be a need to maintain separate file systems for each application to store the user's data, whereas in a

Notes

DBMS system, there would only be one centralised location where data could be down streamed to the different software as and when needed.

2. Records inconsistency

   As each application has the same set of details, changes performed by one user in one utility do not update modifications in other applications when utilising typical file gadget storage. Contrarily, with DBMS structures, there is a single repository of data that is described only once, is accessed by numerous users and the data is consistent.

3. Data sharing

   The main benefit of database control systems is fact sharing. Customers and packages can share statistics with numerous apps and users thanks to DBMS technology. A software locking mechanism may be present that prevents the same piece of data from being changed by humans at the same time as the statistics are stored on one or more servers in the community. Even though the recording equipment lacks this ability.

4. Information searching

   In DBMS structures, searching for and obtaining facts may be quite seamless. As is the case with a standard file-based approach, there is no longer a need to write distinct packages for each search. With DBMS, we can create short queries to check for a few statistics from the stats from db servers at once. The fifth is statistics protection.

   DBMS systems offer a strong framework to safeguard the privacy and security of information. The DBMS ensures that only legitimate users have access to information and that there is a means to define access privileges.

5. Information concurrency

   In DBMS, information is stored on one or more servers connected to a network and a software locking mechanism prevents the same set of information from being changed by two people at the same time.

6. Records integration

   Statistical integration is a method of merging data from several sources to give the user a uniform view of the data. DBMS systems make it possible to integrate data with great ease.

7. When searching for particularly specific information that may be required in the context of a business emergency using a normal file-based technique, it may take hours, whereas DBMS decreases this time to 3 seconds. This is a great advantage of DBMS since we can write short queries to search the database on your behalf and thanks to its built-in searching operations, it will return the records as quickly as possible.

8. Selection making

   Sharing of statistics has advanced and information is now better regulated, enabling company to make smart decisions that will help the firm grow.

9. Records backup and restoration

   Another benefit of DBMS is that it provides a strong foundation for data backup; users no longer need to manually backup their data on a regular basis because DBMS will take care of it automatically. In addition, DBMS restores the database to its previous state in the event of a server crash.

10. Records migration

    There are some information that are accessed rather regularly and others that aren't. Hence, DBMS offers the capacity to access commonly used data as quickly as possible.

Notes

11. Low preservation price

    Even though a DBMS structure is likely to be expensive at the time of purchase, maintaining one costs very little.

12. Records loss

    With DBMS, it is almost unavoidable; provided we don't witness the apocalypse, one could store statistics for thousands of years. As information is safe and has a relatively low market value today compared to generations before, there is no risk of information loss.

5.1.1 Multilevel Transactions

In a database system, multilevel transactions are those that traverse several tiers of abstraction or levels of the hierarchy. Different levels of granularity operations, such as high-level operations on a whole database or lower-level operations on specific records or fields, may be involved in these transactions. Multilevel or hierarchical database designs are frequently linked to the idea of multilevel transactions.

A kind of open-nested transactions known as multi-level transactions has subtransactions that represent actions at several tiers of a tiered system design. Utilising the semantics of high-level operations to boost concurrency is the goal of multi-level transactions. For instance, two "deposit" operations (transactions made on behalf of two funds) on a bank account may be permitted concurrently since they are commutative. But carrying out these high-level actions in parallel necessitates the resolution of potential low-level conflicts (on data pages or indexes, for example) by a low-level synchronisation mechanism.

Page latches, which are inexpensive semaphores held while a page is visited, are typically used to perform this low-level synchronisation in relational database management systems (DBMSs) where records do not span pages. The basic latching approach is not practical for sophisticated DBMSs with intricate high-level operations that might access numerous pages in a dynamically determined (i.e., non-pre-defined) order since it cannot guarantee the indivisibility of random multi-page update operations. High-level operations must instead be carried out as subtransactions that are handled at the lower level by a general concurrency control mechanism. This principle guarantees that the semantic concurrency control at the top level is independent of conflicts at lower levels and can be applied to an arbitrary number of levels.

We handle multi-level transaction management in sophisticated database management systems (DBMSs) that oversee intricate objects by implementing it at the subsequent two tiers:

- Semantic locks are dynamically acquired at object level I L1 and maintained until end-of-transaction (EOT) in compliance with the rigorous two-phase locking protocol. The semantics of the high-level operations are exploited in the lock modes and the lock mode compatibility table, which is in turn derived from the commutativity qualities or semantic compatibility of the operations.
- Page locks at the page level LO are dynamically obtained during a subtransaction's execution and released at the conclusion of the subtransaction (EOS). Note that, unlike the locks of traditional layered transactions, the locks of a subtransaction are not inherited by the parent. The reason multi-level transaction management provides more concurrency than single-level protocols is precisely because it releases the low-level locks as soon as feasible while keeping only a semantically richer lock at a higher level.

Notes

The figure below provides an example of two multi-level transactions being executed in parallel (correctly). Let's say you have an office document filing system where documents can be many pages long and have a complicated structure. Certain high-level procedures, including (1) "changing the font of all instances of a particular component type (e.g., text paragraphs)" and (2) "changing the contents of a figure," are used by users to alter documents. Although these two Change operations on the same document are commutative, there may be potential conflicts at the lower level that need to be resolved because they may access numerous subobjects of the document (for example, when the document's layout is recomputed). This is accomplished in Figure 1 by obtaining locks on the underlying pages that are released at the conclusion of Tll, T12 and T21, respectively, subtransactions.

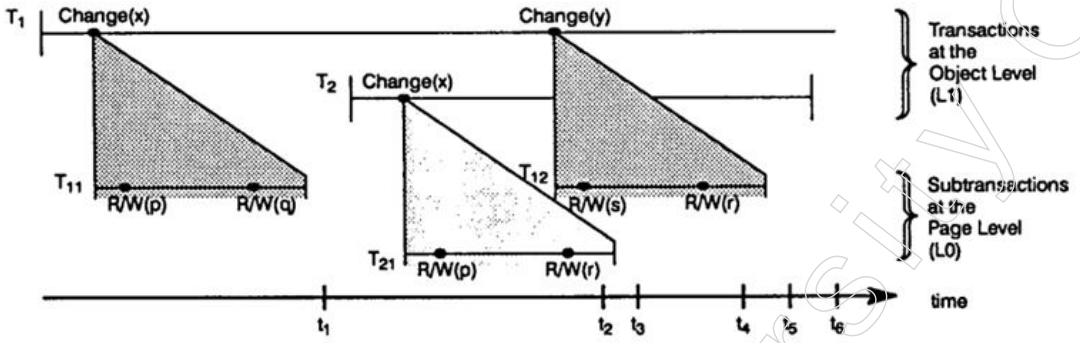


Figure: Parallel execution of two multi-level transactions

Large volumes of derived data are present in sophisticated business applications, where similar occurrences occur. In foreign exchange transactions, a forward transaction (such as a currency swap) would need to calculate a lot of future positions in order to assess risk. For example, it might need to calculate how much Japanese Yen a bank will have on a given date. If subtransactions that release low-level locks early update the derived data, the possibility of data contention in such an application can be minimised.

An essential consequence of multi-level locking is that transactions can no longer be undone using basic state-oriented recovery methods at the page level. Resolved subtransactions require inverse high-level operations to make up for page locks that were released at EOS. The aforementioned operations are then carried out as compensating subtransactions. In the aforementioned figure, two inverse Change operations on y and x, or two extra subtransactions that make up for the completed subtransactions T12 and Tll (reversing the original subtransactions' order), would be needed to reverse transaction T1.

Subtransaction compensation is required for managing aborted transactions as well as crash recovery following a system failure. Regular subtransactions and compensatory subtransactions must both be atomic as a necessary condition. If not, the database state throughout the crash recovery process might not be consistent enough to carry out the required high-level undo operations.

For instance, dangling pointers could be present in a complicated object's storage structures, or certain derived data might only partially reflect the original revisions. To ensure subtransaction atomicity, a low-level recovery mechanism at the page level is required if a subtransaction updates several pages (Figure above). This problem is problematic in that a simplistic implementation of multi-level recovery may produce unnecessary logging and could thus undermine the benefits of the higher concurrency of multi-level transactions.

Here are key points related to multilevel transactions:

Notes

Hierarchy Levels:

- Data is arranged in a hierarchical framework with varying degrees of abstraction in a multi-level database architecture or hierarchical database.
- The database level, schema level, table level and record level are examples of levels in the hierarchy.

Transactional Operations:

- Operations that influence data at different levels of the hierarchy are included in multilevel transactions.
- A transaction could, for instance, entail making changes to the database schema, changing records across several tables and carrying out global activities that impact the entire database.

Atomicity, Consistency, Isolation, Durability (ACID) Properties:

- In order to maintain the database's dependability and consistency, multilevel transactions must follow the ACID specifications.
- The entire transaction is handled as a single, indivisible unit thanks to atomicity. The database is kept up to date both before and after the transaction thanks to consistency. Interference between concurrent transactions is avoided through isolation. Durability makes ensuring that even in the case of failures, the modifications made by the transaction remain in effect.

Transaction Management:

- The coordination and carrying out of multilevel transactions are handled by transaction management systems.
- To preserve consistency, the system has to make sure that either every operation in a multilevel transaction gets applied, or none at all.

Two-Phase Commit (2PC):

- The Two-Phase Commit protocol can be used in distributed database systems with multilevel transactions to coordinate transactions between nodes or levels.
- The protocol guarantees consensus between all involved nodes over whether to commit or cancel the transaction.

Concurrency Control:

- Issues with concurrent access by several transactions must be addressed via Multilevel Transactions.
- Locking and timestamp-based protocols are examples of concurrency control technologies that assist in managing the concurrent execution of transactions to avoid conflicts.

Granularity Control:

- To strike a balance between consistency and efficiency, granularity control entails determining the proper amount of granularity for individual transactions.
- While coarse-grained transactions can affect entire tables or databases, fine-grained transactions can work on individual records.

Notes

Recovery and Logging:

- In the case of failures, recovery mechanisms—such as transaction logging—are essential for getting the database back to a consistent state.
- Logs document modifications made by transactions, enabling the database to be rebuilt in the event of a crash.

5.1.2 Long-lived Transactions (Saga)

A design pattern called long-lived transactions, or sagas, is used in distributed systems to manage and synchronise a sequence of connected and dispersed transactions over a long period of time. When standard short-lived transactions may not be appropriate, the saga pattern is used to preserve data consistency and reliability.

A design pattern known as the SAGA pattern is a series of interconnected smaller transactions. Data consistency is managed and kept across numerous microservices using this technique. A single service completes each transaction and it broadcasts changes in state to other services participating in the Saga. By offering an alternate strategy to the conventional ACID transactions model—which might not be practical in a distributed environment—it contributes to the maintenance of data consistency.

The term "SAGA" refers to the idea of a distributed transaction that is a long story with numerous sections. Each story point in a SAGA is a local transaction and when combined, they tell the entire tale.

It is in charge of overseeing the entire transaction and organising the corrective measures needed in the event that something goes wrong. When handling intricate business procedures involving several services, such order processing, shipping and invoicing, it is helpful. In these situations, coordinating the completion of several transactions is frequently required to preserve data consistency amongst the participating services.

Features of SAGA Pattern

1. Coordinated Transactions: Transactions involving several services or processes can be coordinated with the help of the SAGA pattern. The pattern outlines a series of actions that must be carried out in a coordinated way in order to finish the transaction. Each action may entail a different service or activity.
2. Compensation and Rollback: In the event that a step fails, the SAGA pattern has a method for compensating or rolling back the transaction. This method makes sure that even in the event that one or more steps fail, the transaction stays consistent.
3. Distributed Transactions: Across several services or processes, distributed transactions are supported by the SAGA design. The pattern offers a method for consistently coordinating the transaction across many services or processes.
4. Asynchronous Processing: More concurrency and performance are possible with the SAGA pattern's support for asynchronous processing. This is particularly crucial in distributed systems since different services or processes may process information differently.
5. Handling Errors: A standardised method for addressing problems that arise during a transaction is offered by the SAGA pattern. The pattern makes guarantee that all services or processes participating in the transaction handle problems in the same way.
6. Scalability: The SAGA pattern can accommodate complicated, large-scale transactions involving numerous services or procedures. The pattern offers a means of decomposing

Notes

the transaction into more manageable, smaller phases that can be carried out concurrently across many services or processes.

Example:

Let's examine a practical example of the SAGA pattern in action. Imagine an online shopping application that enables customers to make purchases. Following the placement of an order, the seller must verify and reserve their inventory, process the payment and then dispatch the goods. The entire order needs to be reversed if any of these stages don't work.

For each of the aforementioned functionalities, we can build a different service in order to apply the pattern. It must be planned so that every single one of these transactions happens in unison. Such a website should typically follow this flow:

- Begin Order: Make a new order in the database to begin the order tale.
- Process Payment: Attempt to charge the user's debit card/credit card. If it works, record the order in the database as paid. If not successful, reverse any earlier actions and cancel the order.
- Check Inventory: Verify whether the order's contents are in stock. If not, reverse any earlier actions and cancel the order.
- Reserve Inventory: Put the goods in the order on reserve. If not successful, reverse any earlier actions and cancel the order.
- Ship Order: In the database, mark the order as shipped.

It is clear that every step depends on the one before it. Every stage would be carried out as a distinct service. A successful completion of a step would send a message to the following step. A step that fails would signal the one before it to undo any modifications that have already been done.

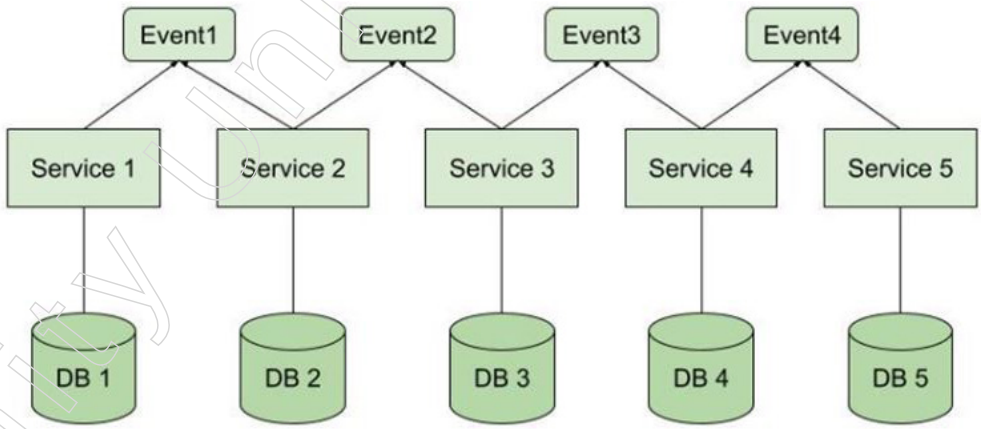


Figure: A typical architecture for the SAGA pattern.

Implementation:

An outline of how the SAGA pattern can be applied to the flow is provided below:

1. Begin Order: Create a new instance of the saga and store it in a database whenever a new order is started. All the information needed to complete the order process, including the user's details and the order details, should be included in the saga instance. Making a new order in the database is the first stage in the story.
2. Process Payment: Update the saga instance to reflect the order as paid in the database if the payment is successful. The story should initiate a compensation step to cancel the order and undo any earlier actions if the payment fails.

Notes

3. Check Inventory: The third step is to verify that all of the items in the order are in stock after the payment has been successfully handled. The story should initiate a compensation step to cancel the order and reverse any earlier actions if any items are out of stock.
4. Reserve Inventory: The inventory for the order is reserved if every item is available. The saga should start a compensation step to cancel the order and reverse any earlier actions if the inventory cannot be reserved for any reason.
5. Ship Order: Shipping the order is the last action in the story. Mark the order as shipped in the database if the order has been paid for, all items are available and the inventory has been reserved. The story should initiate the proper compensation processes to cancel the order and reverse any earlier actions if any of the preceding steps are unsuccessful.

The many stages of the order process across several services or processes are coordinated through the usage of events. When a payment is successfully handled, for example, the service handling the payment may post an event. The service that is in charge of marking the order as paid may use this event and then post an order-paid event. This event can be consumed by the service responsible for reserving the inventory, which can publish an inventory-reserved event. The service that ships the order may use this event, publishing an order shipped event to change the order status in the database.

5.1.3 Data Warehousing

Building and using a data warehouse is known as data warehousing. Data from several heterogeneous sources is combined to create a data warehouse, which facilitates analytical reporting, organised and/or ad hoc searches and decision-making. Data consolidation, data integration and data cleaning are all components of data warehousing.

Data warehouses (DWHs) are repositories where organisations store data electronically by separating it from operational systems and making it accessible for ad-hoc searches and scheduled reporting. In contrast, creating a data warehouse requires creating a data model that can produce insights quickly.

When compared to data found in the operational environment, DWH data is different. To make daily operations, analysis and reporting easier, it is set up such that pertinent data is grouped together. This aids in identifying long-term trends and enables users to make plans based on that knowledge. Thus, it is important for firms to use data warehouses.

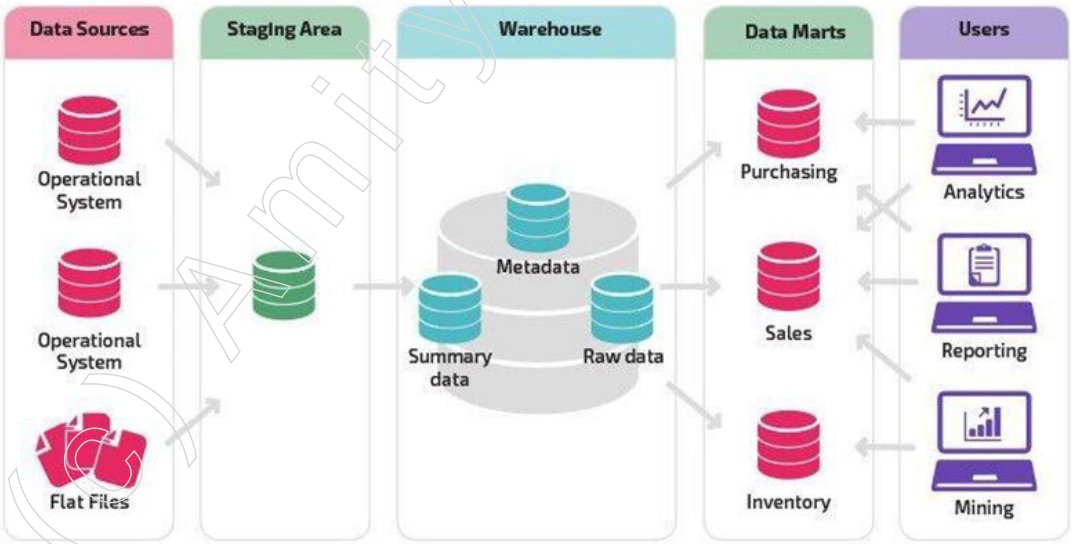


Figure: Data Warehouse Architecture

Notes

In the early stages, four significant factors drove many companies to move into data warehousing:

- Fierce competition
- Government deregulation
- Need to revamp internal processes
- Imperative for customised marketing

The first industries to use data warehousing were telecommunications, finance and retail. Government deregulation in banking and telecoms was primarily to blame for that. Because of the increased competitiveness, retail businesses have shifted to data warehousing. As that industry was deregulated, utility firms joined the organisation. Enterprises in the financial services, healthcare, insurance, manufacturing, pharmaceuticals, transportation and distribution sectors made up the second wave of companies to enter the data warehousing market.

Today, investment on data warehouses is still dominated by the banking and telecoms sectors. Data warehousing accounts for up to 15% of these sectors' technological budgets. Businesses in these sectors gather a lot of transactional data. Such massive amounts of data can be converted through data warehousing into strategic information valuable for decision making.

The majority of global firms were the only ones who used data warehousing in its early phases. Building a data warehouse was expensive and the available tools were not quite sufficient. Only big businesses have the money to invest in the new paradigm. Now that smaller and medium-sized businesses can afford the cost of constructing data warehouses or purchasing turnkey data marts, we are starting to notice a strong presence of data warehousing in these businesses. Examine the database management systems (DBMSs) you have previously employed. The database vendors have now included tools to help you create data warehouses using these DBMSs, as you will discover. The cost of packaged solutions has decreased and operating systems are now reliable enough to support data warehousing operations.

Although earlier data warehouses focused on preserving summary data for high-level analysis, different businesses are increasingly building larger and larger data warehouses. Companies may now collect, purify, maintain and exploit the enormous amounts of data produced by their economic activities. The amount of data stored in data warehouses is now in the terabyte scale. In the telecommunications and retail industries, data warehouses housing many terabytes of data are not unusual.

Take the telecommunications sector, for instance. In a single year, a telecoms operator produces hundreds of millions of call-detail transactions. The corporation must examine these specific transactions in order to market the right goods and services. The company's data warehouse must keep data at the most basic degree of detail.

Consider a chain of stores with hundreds of locations in a similar way. Each store produces thousands of point-of-sale transactions each day. Another example is a business in the pharmaceutical sector that manages thousands of tests and measures to obtain government product approvals. In these sectors, data warehouses are frequently very vast.

Multiple Data Types

If you're just starting to develop your data warehouse, you might only include numeric data. You will quickly understand that that simply including structured numerical data is insufficient. Be ready to take into account more data kinds.

Notes

Companies have historically included structured data, primarily quantitative data, into their data warehouses. From this vantage point, decision support systems might be separated into two groups: knowledge management dealt with unstructured data, while data warehousing dealt with organised data. This line between them is getting fuzzier. For instance, the majority of marketing data is structured data presented as numerical numbers.

Unstructured data in the form of pictures is also present in marketing data. Let's imagine that a decision-maker is conducting research to determine the most popular product categories. During the course of the analysis, the decision maker settles on a particular product type. In order to make additional judgments, he or she would now like to see pictures of the products in that type. How is it possible to do this? Companies are learning that their data warehouses need to integrate both structured and unstructured data.

What kinds of data fall under the category of unstructured data? The data warehouse must incorporate a variety of data kinds to better support decision-making, as seen in the figure below.

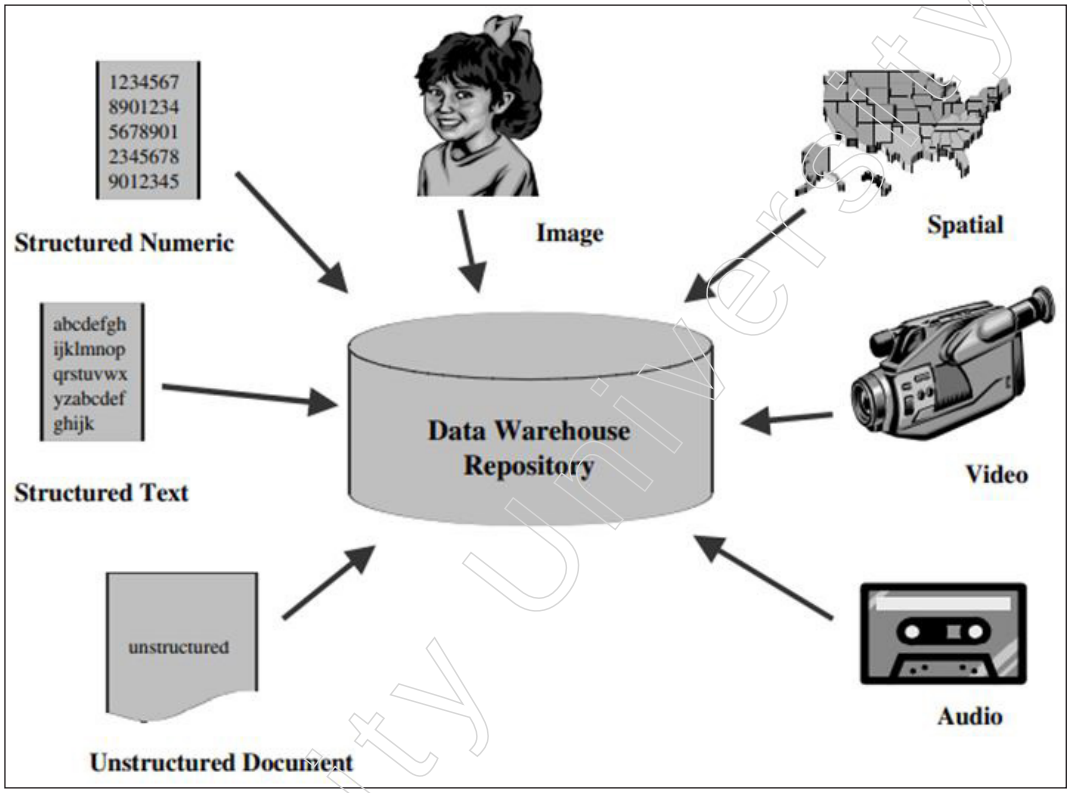


Figure: Data warehouse: multiple data types.

Adding Unstructured Data: The incorporation of unstructured data, particularly text and images, is being addressed by some suppliers by treating such multimedia data as merely another data type. These are classified as relational data and are kept as binary large objects (BLOBs) with a maximum size of 2 GB. These are defined as user-defined types using user-defined functions (UDFs) (UDTs).

It is not always possible to store BLOBs as just another relational data type. A server that can send several streams of video at a set rate and synchronise them with the audio component, for instance, is needed for a video clip. Specialised servers are being made available for this purpose.

Searching Unstructured Data: Your data warehouse has been improved by the addition of unstructured data. Do you have any other tasks to complete? Of course, integrating

Notes

such data is largely useless without the capacity to search it. In order to help users find the information they need from unstructured data, vendors are increasingly offering new search engines. An illustration of an image search technique is querying by image content.

Preindexing photographs based on forms, colours and textures is possible with this product. The chosen images are shown one after the other when more than one image matches the search criteria.

Retrieval engines preindex the textual documents for free-form text data in order to support word, character, phrase, wild card, proximity and Boolean searches. Some search engines are capable of searching and replacing words with their equivalents. The word mice will also turn up in documents when you search for it.

Directly searching audio and video data is still a research topic. These are typically defined using free-form language, which is then searched using the existing textual search techniques.

Searching audio and video data directly is still in the research stage. Usually, these are described with free-form text and then searched using textual search methods that are currently available.

Spatial Data: Imagine one of your key users—perhaps the Marketing Director—is online and accessing your data warehouse to conduct an analysis. Show me the sales for the first two quarters for all products compared to last year in shop XYZ, requests the Marketing Director. He or she considers two additional questions after going over the findings. What is the typical income of those who reside in the store's neighbourhood? How far did those people drive to get to the store on average? These queries can only be addressed if spatial data is included in your data warehouse.

Your data warehouse's value will increase significantly with the addition of spatial data. Examples of spatial data include an address, a street block, a city quarter, a county, a state and a zone. Vendors have started addressing the requirement for spatial data. In order to combine spatial and business data, some database providers offer spatial extenders to their products via SQL extensions.

Data mining is a technique used with the Internet of Things (IoT) and artificial intelligence (AI) to locate relevant, possibly useful and intelligible information, identify patterns, create knowledge graphs, find anomalies and establish links in massive data. This procedure is crucial for expanding our understanding of a variety of topics that deal with unprocessed data from the web, text, numbers, media, or financial transactions.

By fusing several data mining techniques for usage in financial technology and cryptocurrencies, the blockchain, data sciences, sentiment analysis and recommender systems, its application domain has grown. Additionally, data mining benefits a variety of real-world industries, including biology, data security, smart grids, smart cities and maintaining the privacy of health data analysis and mining. Investigating new developments in data mining that incorporate machine learning methods and artificial neural networks is also required.

Machine learning and deep learning are undoubtedly two of the areas of artificial intelligence that have received the most research in recent years. Due to the development of deep learning, which has provided data mining with previously unheard-of theoretical and application-based capabilities, there has been a significant change over the past few decades. The articles in this Topic will examine both theoretical and practical applications of knowledge discovery and extraction, image analysis, classification and clustering, as well as FinTech and cryptocurrencies, the blockchain and data security, privacy-preserving data

mining and many other topics. Both theoretical and practical model-focused contributions are sought. According to their formal and technical soundness, experimental support and relevance, papers will be chosen for inclusion.

Notes

According to Fayadd, Piatesky-Shapiro and Smyth (1996), data mining (DM) refers to a collection of particular techniques and algorithms created specifically for the purpose of identifying patterns in unprocessed data. The massive amount of data that must be managed more easily in sectors like commerce, the medical industry, astronomy, genetics, or banking has given rise to the DM process. Additionally, the exceptional success of hardware technologies resulted in a large amount of storage capacity on hard drives, which posed a challenge to the emergence of numerous issues with handling enormous amounts of data. Of course, the Internet's rapid expansion is the most significant factor in this situation.

The core of the DM process is the application of methods and algorithms to find and extract patterns from stored data, although data must first be pre-processed before this stage. It is well known that DM algorithms alone do not yield satisfactory results. Finding useful knowledge from raw data thus requires the sequential application of the following steps: developing an understanding of the application domain, creating a target data set based on an intelligent way of selecting data by focusing on a subset of variables or data samples, data cleaning and pre-processing, data reduction and projection, choosing the data mining task, choosing the data mining algorithm, the data mining step and interpreting the results.

Regression is learning a function that maps a data item to a real-valued prediction variable. Clustering is the division of a data set into subsets (clusters). Association rules determine implication rules for a subset of record attributes. Summarisation involves methods for finding a compact description for a subset. Classification is learning a function that maps a data item into one of several predefined classes .

The DM covers a wide range of academic disciplines, including artificial intelligence, machine learning, databases, statistics, pattern identification in data and data visualisation. Finding trends in the data and presenting crucial information in a way that the average person can understand it are the main objectives here. For ease of usage, it is advised that the information acquired be simple to understand. Getting high-level data from low-level data is the overall goal of the process.

The DM method can be applied in a wide range of scientific fields, including biology, medicine, genetics, astronomy, high-energy physics, banking and business, among many others. DM algorithms and approaches can be used on a variety of information, including plain text and multimedia formats.

5.1.4 Data Mining

Gregory Piatesky-Shapiro, a researcher, first coined the phrase "data mining" (DM) in 1989. At the time, there weren't many data mining tools available for completing a single problem. The C4.5 decision tree technique, the SNNS neural network and parallel coordinate visualisation are among examples (Quinlan, 1986). (Inselberg, 1985). These techniques required significant data preparation and were challenging to use.

Suites, or second-generation data mining systems, were created by manufacturers beginning in 1995. These tools considered the fact that the DM process necessitates several kinds of data analysis, with the bulk of the work occurring during the data cleaning and preprocessing stages. Users may carry out a number of discovery activities (often classification, clustering and visualisation) using programmes like SPSS Clementine,

Notes

SGI Mineset, IBM Intelligent Miner, or SAS Enterprise Miner. These programmes also offered data processing and visualisation. A GUI (Graphical User Interface), invented by Clementine, was one of the most significant developments since it allowed users to construct their information discovery process visually.

In 1999, more than 200 tools were available for handling various tasks, but even the best of them only handled a portion of the whole DM architecture. Preprocessing and data cleaning were still required. Data-mining-based "vertical solutions" have emerged as a result of the growth of these types of applications in industries like direct marketing, telecommunications and fraud detection. The systems HNC Falcon for credit card fraud detection, IBM Advanced Scout for sports analysis and NASD DM Detection system are the best examples of such applications.

In recent years, parallel and distributed computers were used as approaches to the DM process. These instructions caused Parallel DM and Distributed DM to appear. Data sets are distributed to high performance multi-computer machines for analysis in Parallel DM. All algorithms that were employed on single-processor units must be scaled in order to run on parallel computers, despite the fact that these types of machines are becoming more widely available. The Parallel DM technique is appropriate for transaction data, telecom data and scientific simulation. Distributed DM must offer local data analysis solutions as well as global methods for recombining local results from each computing unit without requiring a significant amount of data to be transferred to a central server. Grid technologies combine distributed parallel computing and DM.

In general, "mining" refers to the process of extracting a valuable resource from the earth, such as coal or diamonds. Knowledge mining from data, knowledge extraction, data/pattern analysis, data archaeology and data dredging are all terms used in computer science to describe data mining. It is essentially the technique of extracting valuable information from a large amount of data or data warehouses. It's clear that the term itself is a little perplexing.

The product of the extraction process in coal or diamond mining is coal or diamond. The result of the extraction process in Data Mining, however, is not data!! Instead, the patterns and insights we get at the end of the extraction process are what we call data mining outcomes. In that respect, Data Mining can be considered a step in the Knowledge Discovery or Knowledge Extraction process.

The phrase "Knowledge Discovery in Databases" was coined by Gregory Piatetsky-Shapiro in 1989. The term 'data mining,' on the other hand, grew increasingly prevalent in the business and media circles. Data mining and knowledge discovery are terms that are being used interchangeably.

Data Mining and Neural Networks

The core of the DM process is the data mining step. Neural networks are utilised to complete several DM jobs. Since the human brain serves as the computing model for these networks, neural networks should be able to learn from experience and change in reaction to new information. Neural networks can find previously unidentified links and learn complicated non-linear patterns in the data when subjected to a collection of training or test data.

Our brain's capacity to differentiate between two items is one of its most crucial abilities. This capability of differentiating between what is friendly and what is dangerous for particular species has allowed numerous species to evolve successfully.

Regression is a learning process that converts a specific piece of data into a genuine variable that can be predicted. This is comparable to what individuals actually do: after

Notes

seeing a few examples, they can learn to extrapolate from them in order to apply the information to unrelated issues or situations. One of the benefits of adopting neural network technology is its capacity for generalisation.

The Hebbian learning theory, which holds that information is stored by associations with relevant memories, was validated by neurophysiologists. A intricate network of concepts with semantically similar meanings exists in human brain. According to the hypothesis, when two neurons in the brain are engaged simultaneously, their connection develops stronger and the physical properties of their synapse are altered. The neural network field's associative memory branch focuses on developing models that capture associative activity.

Associative memories with a restricted capacity have been demonstrated in neural networks like Binary Adaptive Memories and Hopfield networks. Different neural models, including back propagation networks, recurrent back propagation networks, self-organising maps, radial basis function networks, adaptive resonance theory networks (ART), probabilistic neural networks and fuzzy perceptions, are used to resolve these DM procedures. Certain DM procedures are available to any structure.

Businesses who were late to implement data mining are rapidly catching up to the others. Making crucial business decisions frequently involves using data mining to extract key information. Data mining is predicted to become as commonplace as some of the current technologies. Among the major trends anticipated to influence the direction of data mining are:

Multimedia Data Mining

This is one of the most recent techniques that is gaining acceptance due to its increasing capacity to accurately capture important data. Data extraction from several types of multimedia sources, including audio, text, hypertext, video, pictures and more, is a part of this process. Afterwards, the extracted data is transformed into a numerical representation in a variety of formats. Using this technique, you may create categories and clusters, run similarity tests and find relationships.

Ubiquitous Data Mining

With this technique, data from mobile devices is mined to obtain personal information about people. Despite having a number of drawbacks of this kind, including complexity, privacy, expense and more, this approach is poised to develop and find use in a variety of industries, particularly when researching human-computer interactions.

Distributed Data Mining

Since it includes the mining of a sizable amount of data held across numerous corporate sites or within multiple companies, this type of data mining is becoming more and more popular. To collect data from various sources and produce improved insights and reports based on it, highly complex algorithms are used.

Spatial and Geographic Data Mining

This is a newly popular sort of data mining that involves information extraction from environmental, astronomical and geographic data, including photographs captured from space. This kind of data mining shows a variety of factors, mostly employed in geographic information systems and other navigation applications, such as distance and topology.

Time Series and Sequence Data Mining

The analysis of cyclical and seasonal trends is the main application of this kind of data

Notes

mining. Even random incidents that happen outside of the usual course of events can be studied with the use of this approach. Retail businesses mostly employ this strategy to analyse consumer behaviour and purchase habits.

Data Mining on Databases

Steps of Knowledge Discovery Database Process

Step 1. Data Collection

You require some data for any research. The research team locates pertinent tools to extract, convert and process data from various websites or big data settings like data lakes, servers, or warehouses. Both structured and unstructured files could be present. Data migration issues and the database's inability to demonstrate file format compliance might occasionally create errors.

To avoid incorrect or erroneous data entry in this case, the errors should be eliminated at the point of entry.

Step 2: Preparing Datasets

Investigate datasets after collection. As part of pre-processing, create their profiles. It is the transformation step, in fact. All records are cleaned up, kept consistent and abnormalities are eliminated here. Resources are provided in this manner for efficient data analysis. The final phase would then be the error-fixing, which is done in the next step. The method is used by Eminenture's data mining specialists.

Step 3: Cleansing Data

Here, cleaning refers to eliminating noise and duplication from data collecting. Noises in the database relate to duplicates, corrupt, incomplete and other anomalies. Following are the subsets for tidy and clean databases:

- De-duplication
- Data appending
- Normalisation
- Typos removal
- Standardisation

Step 4: Data Integration

This procedure necessitates synthesising information from several sources. A variety of apps and technologies for migration and synchronisation are used in the knowledge discovery (KDD) process.

This procedure mainly entails extracting, loading and changing datasets. Additionally known as the ETL procedure. Information needed for study must be:

- Extracted
- Transformed
- Loaded for deep analysis

Step 5: Data Analysis

Analysis is the process of delving deeply into insights. Datasets are processed here by filtering. They are carefully scrutinised to see whether they can be helpful and the ideal fit for these processes:

Notes

- Neural network
- Decision trees
- Naïve Byaes
- Clustering
- Association
- Regression, etc.

Step 6: Data Transformation

- Using data mapping, elements are assigned from sources to destinations.
- OCR conversion for scanning, recognition and translation of data
- Coding to convert all of the data to the same format.

In this process, data is transformed into a format and structure that will be exactly the same for further data mining. Sometimes, PDFs or data in many formats are available. It is essential to digitise them. As a result, the following actions are taken:

Step 7: Modeling or Mining

This is the most important step, which uses scripts and apps to extract pertinent patterns to support goals.

- Data transformation into models or patterns is covered.
- Verification is used to check the veracity of facts.

Step 8: Validating Models

This KDD process stage, also known as evaluation, includes validating found patterns. The data scientist creates a few predictive models (or projections) to evaluate whether they can deliver the desired outcomes. By employing classification and characterisation techniques or the best-fit method, validation demonstrates that the models accurately fit the predicted models.

- Finding each model's interestingness score or rating is a step in this approach.
- An overview of the Pan database.
- visualisation for simple comprehension and analysis.

Step 9: Knowledge Presentation

As the name implies, you must show your discoveries in extensive datasets throughout this stage of the data mining process. It can be made simpler by using visualisation tools like Data Studio.

- Making reports
- Establish classification, characterisation and other types of discriminatory norms.

Step 10: Execution

Finally, numerous applications or machine learning are used with the driven decisions or findings. If done correctly, it can be used to automate operations using artificial intelligence (AI).

Data Mining Functionalities

Six Basic Data Mining Activities

It's critical to distinguish between the activities involved in data mining, the processes

Notes

by which they are carried out and the methodologies that make use of these activities to find business prospects. The most important techniques can essentially be reduced to six tasks:

Clustering and Segmentation

The process of clustering entails breaking up a huge collection of items into smaller groups that share specific characteristics (Figure below). The distinction between classification and clustering is that, in the clustering task, the classes are not predefined. Instead, the decision or definition of that class is based on the evaluation of the classes that occurs once clustering is complete.

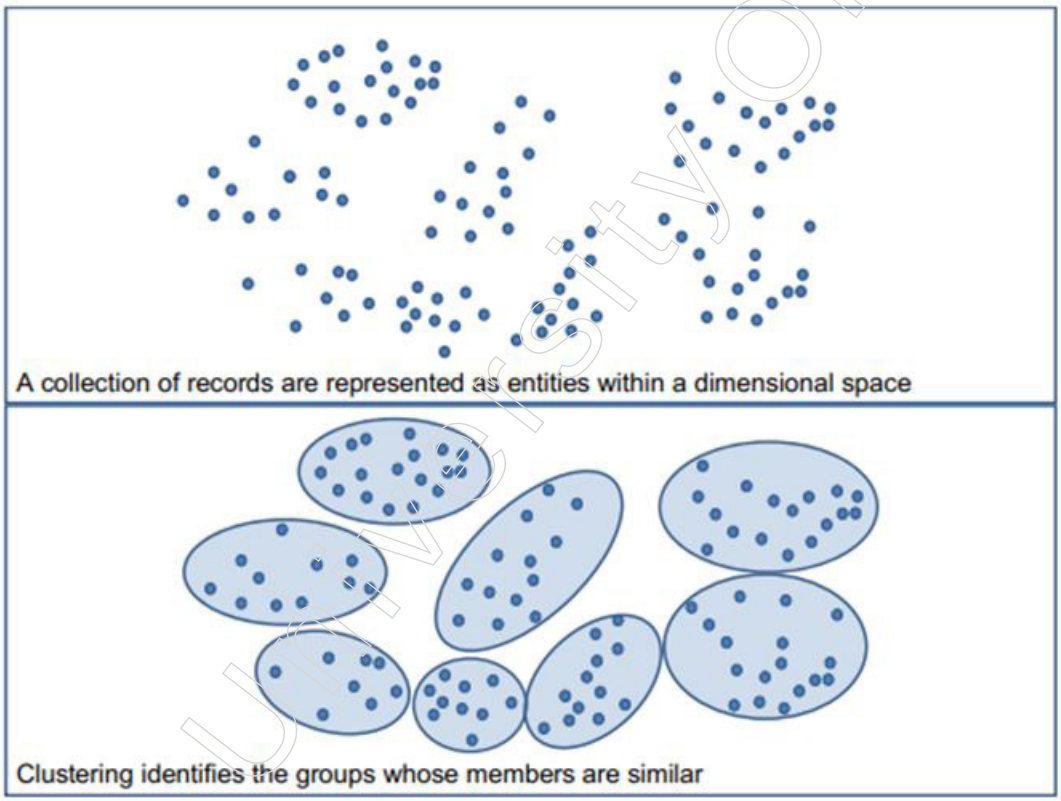


Figure: Example of Clustering

When you need to separate data but are unsure exactly what you are looking for, clustering can be helpful. To check if any relationships can be deduced through the clustering process, you might, for instance, want to assess health data based on particular diseases together with other variables. Clustering can be used to identify a business problem area that needs more investigation in conjunction with other data mining operations. As an illustration, segmenting the market based on product sales before examining why sales are low within a certain market segment.

The clustering method arranges data instances so that each group stands out from the others and that its members are easily distinguishable from one another. Since there are no established criteria for classification, the algorithms essentially "choose" the features (or "variables") that are used to gauge similarity. The records in the collection are arranged according to how similar they are. The results may need to be interpreted by someone with knowledge of the business context to ascertain whether the clustering has any particular meaning. In some cases, this may lead to the culling out of variables that do not carry meaning or may not be relevant, in which case the clustering can be repeated in the absence of the culled variables.

Notes

Classification

People who categorise the world into two groups and those who do not are divided into two groups. But truly, it's human nature to classify objects into groups based on a shared set of traits. For instance, we classify products into product classes, or divide consumer groups by demographic and/or psychographic profiles (e.g., marketing to the profitable 18 to 34-year-olds).

The results of identified dependent variables can be used for classification when segmentation is complete. The process of classifying data into preset groups is called classification (Figure below). These classifications might either be based on the output of a clustering model or could be described using the analyst's chosen qualities. The programme is given the class definitions and a training set of previously classified objects during a classification process and it then makes an effort to create a model that can be used to correctly categorise new records. For instance, a classification model can be used to categorise customers into specific market sectors, assign meta-tags to news articles depending on their content and classify public firms into good, medium and poor investments.

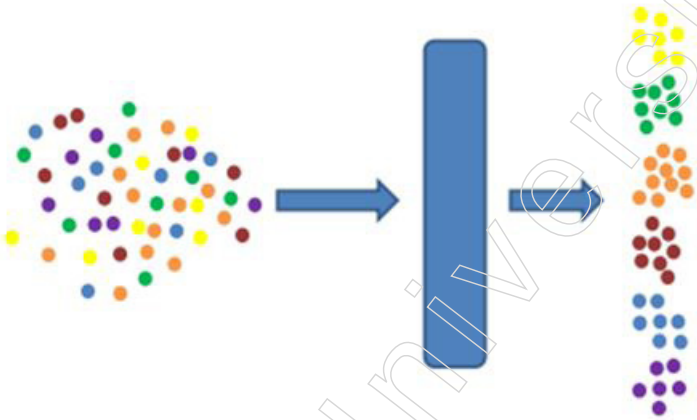


Figure: Classification of records based on defined characteristics.

Estimation

Estimation is the process of giving an object a continuously varying numerical value. For instance, determining a person's credit risk is not always a yes-or-no decision; it is more likely to involve some form of scoring that determines a person's propensity to default on a loan. In the categorisation process, estimation can be employed (for example, in a market segmentation process, to estimate a person's annual pay).

Because a value is being given to a continuous variable, one benefit of estimate is that the resulting assignments can be sorted according to score. Consequently, a ranking method may, for instance, assign a value to the variable "probability of buying a time-share vacation package" and then order the applicants according to that projected score, making those individuals the most likely.

Estimation is usually used to establish a fair guess at an unknowable value or to infer some likelihood to execute an action. Customer lifetime value is one instance where we tried to create a model that reflected the future worth of the relationship with a customer.

Prediction

Prediction is an attempt to categorise items based on some anticipated future behaviour, which is a slight distinction from the preceding two jobs. Using historical data

Notes

where the classification is already known, classification and estimation can be utilised to make predictions by creating a model (this is called training). Then, using fresh data, the model can be used to forecast future behaviour.

When utilising training sets for prediction, you must be cautious. The data may have an inherent bias, which could cause you to make inferences or conclusions that are pertinent to the bias. Utilise various data sets for testing, testing and more testing!

Affinity Grouping

The technique of analysing associations or correlations between data items that show some sort of affinity between objects is known as affinity grouping. For instance, affinity grouping could be used to assess whether customers of one product are likely to be open to trying another. When doing marketing campaigns to cross-sell or up-sell a consumer on more or better products, this type of analysis is helpful. The creation of product packages that appeal to broad market segments can also be done in this way. For instance, fast-food chains may choose particular product ingredients to go into meals packed for a specific demographic (such as the "kid's meal") and aimed at the people who are most likely to buy those packages (e.g., children between the ages of 9 and 14).

Description

The final duty is description, which is attempting to characterise what has been found or to provide an explanation for the outcomes of the data mining process. Another step toward a successful intelligence programme that can locate knowledge, express it and then assess possible courses of action is the ability to characterise a behaviour or a business rule. In fact, we may claim that the metadata linked to that data collection can include the description of newly acquired knowledge.

How Data Mining Works

Data mining is the process of examining and analysing huge chunks of data to discover significant patterns and trends. Numerous applications exist for it, including database marketing, credit risk management, fraud detection, spam email screening and even user sentiment analysis.

There are five steps in the data mining process. Data is first gathered by organisations and loaded into data warehouses. The data is then kept and managed, either on internal servers or on the cloud. The data is accessed by business analysts, management groups and information technology specialists, who then decide how to organise it. The data is next sorted by application software according to the user's findings and ultimately the end-user presents the data in a manner that is simple to communicate, such a graph or table.

Data Warehousing and Mining Software

Based on user requests, data mining tools examine relationships and patterns in data. A business might employ data mining software to produce information classifications, for instance. As an example, consider a restaurant that wishes to use data mining to figure out when to run specific specials. It examines the data it has gathered and establishes classes according to the frequency of client visits and the items they purchase.

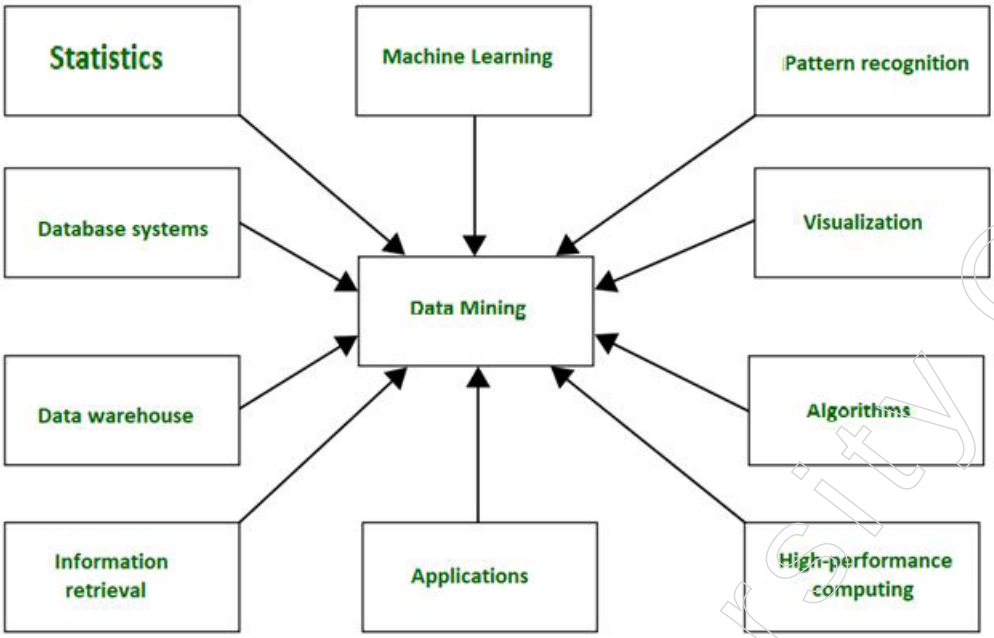
Other times, data miners hunt for information clusters based on logical connections, or they analyse associations and sequential patterns to infer trends in customer behaviour.

Data mining includes warehousing as a crucial component. Companies who warehouse their data into a single database or application. An organisation can isolate specific data segments for analysis and use by particular users using a data warehouse.

In other instances, analysts could start with the data they need and build a data warehouse from scratch using those specifications.

Notes

Main Purpose of Data Mining



In general, data mining has been combined with a variety of other techniques from other domains, such as statistics, machine learning, pattern recognition, database and data warehouse systems, information retrieval, visualisation and so on, to gather more information about the data and to help predict hidden patterns, future trends and behaviours, allowing businesses to make decisions.

Data mining, in technical terms, is the computer process of examining data from many viewpoints, dimensions and angles and categorising/summarising it into useful information.

Data Mining can be used on any sort of data, including data from Data Warehouses, Transactional Databases, Relational Databases, Multimedia Databases, Spatial Databases, Time-series Databases and the World Wide Web.

Data Mining as a Whole Process

The data mining process can be broken down into these four primary stages:

Data gathering: Data that is pertinent to an analytics application is gathered and identified. A data lake or data warehouse, which are becoming more and more popular repositories in big data contexts and include a mixture of structured and unstructured data, may be where the data is located. Another option is to use other data sources. Wherever the data originates, a data scientist frequently transfers it to a data lake for the process' remaining steps.

Data preparation: A series of procedures are included in this stage to prepare the data for mining. Data exploration, profiling and pre-processing come first, then work to clean up mistakes and other problems with data quality. Unless a data scientist is attempting to evaluate unfiltered raw data for a specific application, data transformation is also done to make data sets consistent.

Mining the data: A data scientist selects the best data mining technique after the data is ready and then uses one or more algorithms to perform the mining. Before being applied

Notes

to the entire set of data in machine learning applications, the algorithms are often trained on sample data sets to look for the desired information.

Data analysis and interpretation: Analytical models are developed using the data mining findings to guide decision-making and other business activities. Additionally, the data scientist or another member of the data science team must convey the results to users and business executives, frequently by using data storytelling approaches and data visualisation.

Applications of Data Mining

- Financial Analysis
- Biological Analysis
- Scientific Analysis
- Intrusion Detection
- Fraud Detection
- Research Analysis

Real-life examples of Data Mining

Market Basket Analysis (MBA) is a technique for analysing the purchases made by a client in a supermarket. The idea is to use the concept to identify the things that a buyer buys together. What are the chances that if a person buys bread, he or she will also buy butter?

Protein Folding is a technology that examines biological cells in detail and predicts protein connections and functions within them. This research could lead to the discovery of causes and potential therapies for Alzheimer's, Parkinson's and cancer diseases caused by protein misfolding.

Fraud Detection: In today's world of cell phones, we can utilise data mining to compare suspicious phone activity by analysing cell phone activities. This may aid in the detection of cloned phone calls. Similarly, when it comes to credit cards, comparing recent transactions to previous purchases can help uncover fraudulent behaviour.

Business intelligence, Web search, bioinformatics, health informatics, finance, digital libraries and digital governments are just a few of the successful applications of data mining.

Data Mining Techniques

Algorithms and a variety of techniques are used in data mining to transform massive data sets into useable output. The most often used kinds of data mining methods are as follows:

- Market basket analysis and association rules both look for connections between different variables. As it attempts to connect different bits of data, this relationship in and of itself adds value to the data collection. For instance, association rules would look up a business's sales data to see which products were most frequently bought together; with this knowledge, businesses may plan, advertise and anticipate appropriately.
- To assign classes to items, classification is used. These categories describe the qualities of the things or show what the data points have in common. The underlying data can be more precisely categorised and summed up across related attributes or product lines thanks to this data mining technique.

Notes

- Clustering and categorisation go hand in hand. Clustering, on the other hand, found similarities between objects before classifying them according to how they differ from one another. While clustering might reveal groupings like “dental health” and “hair care,” categorisation can produce groups like “shampoo,” “conditioner,” “soap,” and “toothpaste.”
- Decision trees are employed to categorise or forecast a result based on a predetermined set of standards or choices. A cascading series of questions that rank the dataset based on responses are asked for input using a decision tree. A decision tree allows for particular direction and user input when digging deeper into the data and is occasionally represented visually as a tree.
- An method called K-Nearest Neighbor (KNN) classifies data according to how closely it is related to other data. KNN is based on the idea that data points near to one another have a higher degree of similarity than other types of data. This supervised, non-parametric method forecasts group characteristics from a set of individual data points.
- The nodes of neural networks are used to process data. These nodes have an output, weights and inputs. Through supervised learning, data is mapped (similar to how the human brain is interconnected). This model can be fitted to provide threshold values that show how accurate a model is.
- In order to forecast future results, predictive analysis aims to use historical data to create graphical or mathematical models. This data mining technique, which overlaps with regression analysis, seeks to support an unknown figure in the future based on already available data.

5.1.5 Active Database

For a considerable amount of time, rules that define actions that are automatically triggered by specific occurrences have been seen as significant improvements to database systems. Actually, the idea of triggers—a method for defining specific kinds of active rules—was present in the first drafts of the SQL specification for relational databases and they are currently included in SQL-99 and subsequent standards. Trigger versions vary throughout commercial relational DBMSs, including Microsoft SQLServer, Oracle and DB2. Nonetheless, since the first trigger models were put forth, a great deal of work has been done to determine what a general model for active databases ought to entail.

Generalised Model for Active Databases and Oracle Triggers

The event-condition-action (ECA) model is the one that has been applied to define active database rules. Three elements make up a rule in the ECA model:

1. The event(s) that triggers the rule: Most often, these events are specifically applied database update activities. But they could also be temporal events or other types of external events in the general model.
2. The condition that determines whether the rule action should be executed: Following the occurrence of the triggering event, an optional condition might be assessed. In the absence of a condition, the action will take place as soon as the event takes place. If a condition is stated, it is evaluated first and the rule action is only carried out if the evaluation returns true.
3. The action to be taken: The action is typically a series of SQL statements, but it could also be an automatically running external programme or a database transaction.

Notes

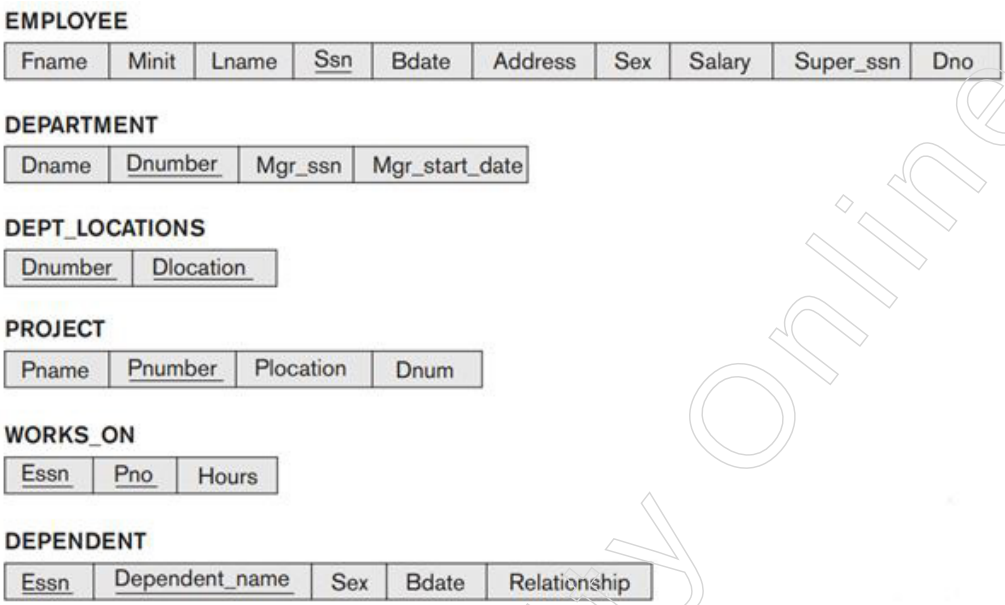


Figure: Schema diagram for the COMPANY relational database schema.

Let's look at a few examples to clarify these ideas. The examples are based on a much simplified variation of the COMPANY database application from Figure above and are shown in Figure below, with each employee having a name (Name), Social Security number (Ssn), salary (Salary), department to which she is currently assigned (Dno, a foreign key to DEPARTMENT) and a direct supervisor (Supervisor_ssn, a (recursive) foreign key to EMPLOYEE). We will assume for the purposes of this example that Dno is allowed to be NULL, meaning that an employee can be momentarily unassigned to any department. Every department has a name (Dname), a number (Dno), a manager (Manager_ssn, which is a foreign key to EMPLOYEE) and the total salary of all employees assigned to the department (Total_sal).

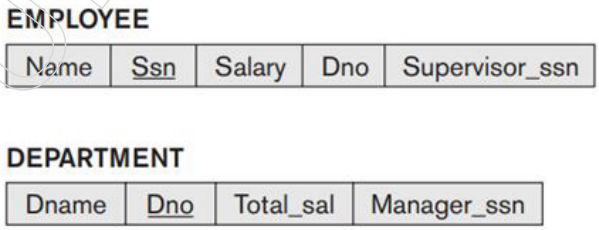


Figure: A simplified COMPANY database used for active rule examples.

It is important to note that the Total_sal attribute is really derived and its value should equal the total salary of all employees assigned to that specific department. An active rule can be used to keep such a derived attribute's value correct. The following events must first be identified as they have the potential to alter the value of Total_sal:

1. Inserting (one or more) new employee tuples
2. Changing the salary of (one or more) existing employees
3. Changing the assignment of existing employees from one department to another
4. Deleting (one or more) employee tuples

If the new employee is immediately allocated to a department, or if the value of the Dno property for the new employee tuple is not NULL (assuming NULL is allowed for Dno), then in the case of event 1, we merely need to recompute Total_sal. Therefore, it would be this

Notes

that has to be examined. To find out whether the employee whose salary is being adjusted (or who is being eliminated) is currently allocated to a department, a similar condition may be tested for events 2 (and 4). There is no need for a condition because, in the case of event 3, we will always take action to preserve the value of Total_sal accurately (the action is always executed).

The automated change of the employee's department's Total_sal value to reflect the newly inserted, updated, or removed employee's salary is the action for events 1, 2 and 4. When it comes to event 3, two steps must be taken: one to update the employee's previous department's Total_sal and another to update the employee's new department's Total_sal.

The Oracle DBMS notation can provide the four active rules (or triggers) R1, R2, R3 and R4—which match to the aforementioned scenario—as illustrated in the following Figure. To demonstrate the syntax for creating triggers in Oracle, let's look at rule R1. Total_sal1 for R1 is the name of the trigger (or active rule) specified by the CREATE TRIGGER statement. The rule will take effect following the occurrence of the circumstances that set it off, as stated in the AFTER clause. The AFTER keyword is followed by the triggering events, which in this case are the addition of a new employee.

```
(a) R1: CREATE TRIGGER Total_sal1
        AFTER INSERT ON EMPLOYEE
        FOR EACH ROW
        WHEN ( NEW.Dno IS NOT NULL )
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal + NEW.Salary
            WHERE Dno = NEW.Dno;

    R2: CREATE TRIGGER Total_sal2
        AFTER UPDATE OF Salary ON EMPLOYEE
        FOR EACH ROW
        WHEN ( NEW.Dno IS NOT NULL )
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal + NEW.Salary - OLD.Salary
            WHERE Dno = NEW.Dno;

    R3: CREATE TRIGGER Total_sal3
        AFTER UPDATE OF Dno ON EMPLOYEE
        FOR EACH ROW
            BEGIN
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal + NEW.Salary
            WHERE Dno = NEW.Dno;
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal - OLD.Salary
            WHERE Dno = OLD.Dno;
            END;
```

Figure: Specifying active rules as triggers in Oracle notation. Triggers for automatically maintaining the consistency of Total_sal

The relation on which the rule is specified—EMPLOYEE for R1—is specified by the ON clause. The rule will be invoked once for each row that is impacted by the triggering event, according to the optional keywords FOR EVERY ROW.

If there are any criteria that must be met after the rule is activated but before the action is carried out, they can be listed in the optional WHEN clause. Lastly, a PL/SQL block containing one or more SQL statements or calls to external procedures is used to specify the action(s) to be taken.

Several characteristics of active rules are exemplified by the four triggers (active rules), R1, R2, R3 and R4. First, the regular SQL update commands—INSERT, DELETE

Notes

and UPDATE—are the fundamental events that can be provided for setting off the rules. In Oracle nomenclature, they are denoted by the keywords INSERT, DELETE and UPDATE. When using UPDATE, one can indicate which characteristics need to be modified, such as by writing UPDATE OF Salary, Dno. Second, the triggering event must provide a means for the rule designer to make reference to the tuples that it has added, removed, or altered. In Oracle notation, the terms NEW and OLD signify freshly entered or updated tuples, respectively, while OLD denotes deleted tuples or tuples that were not modified.

As a result, after applying an INSERT operation to the EMPLOYEE relation, rule R1 is activated. The operation is carried out in R1 if the criterion (NEW.Dno IS NOT NULL) is verified as true, indicating that the recently inserted employee tuple is associated with a department. The newly entered employee's pay (NEW.pay) is added to the Total_sal property of their relevant department, updating the DEPARTMENT tuple(s) associated with them.

Similar to Rule R1, except instead of being triggered by an INSERT, Rule R2 is triggered by a UPDATE transaction that modifies an employee's SALARY. An update to the Dno attribute of EMPLOYEE, which denotes a shift in an employee's departmental assignment, initiates Rule R3. Since R3 doesn't have a condition to verify, the action is carried out whenever the triggering event takes place. The reassigned employees' salaries are added to the new department's Total_sal and subtracted from the old department's Total_sal in order to update both the old and new departments. It should be noted that in the event that Dno is NULL, no department will be chosen for the rule action, therefore this should still function.

The effect of the optional FOR EACH ROW clause, which indicates that the rule is invoked independently for each tuple, should be noted. We refer to this as a rowlevel trigger. The trigger would be referred to as a statement-level trigger and would be triggered once for each triggering statement if this clause was omitted. To observe the difference, have a look at the following update operation, which raises all department 5 employees by 10%. Rule R2 would be triggered by this operation.

```
UPDATE EMPLOYEE
SET Salary = 1.1 * Salary
WHERE Dno = 5;
```

A rule utilising row-level semantics, like R2 in the previous Figure, would be activated once for each row since the above statement might alter many records, whereas a rule using statement-level semantics would only be triggered once. The user can select which of the aforementioned choices to apply to each rule using the Oracle system. A row-level trigger is created when the optional FOR EACH ROW clause is included; a statement-level trigger is created when it is omitted. It should be noted that only row-level triggers can be used with the keywords NEW and OLD.

As an additional illustration, let's say we wish to determine if an employee's pay exceeds that of their immediate supervisor. This rule may be triggered by a number of things, such as hiring a new employee, altering an employee's pay, or switching an employee's supervisor. Assume that calling the external procedure inform_supervisor, 6 will alert the supervisor, is the appropriate course of action. The rule might therefore be expressed as R5 shown in figure below.

Notes

```
R4:  CREATE TRIGGER Total_sal4
     AFTER DELETE ON EMPLOYEE
     FOR EACH ROW
     WHEN ( OLD.Dno IS NOT NULL)
          UPDATE DEPARTMENT
          SET Total_sal = Total_sal – OLD.Salary
          WHERE Dno = OLD.Dno;
```

Figure: Trigger for comparing an employee's salary with that of his or her supervisor.

Design and Implementation Issues for Active Databases

The first problem is the grouping, activation and deactivation of rules. Apart from enabling users to create rules, an active database system should also enable them to utilise their rule names to activate, deactivate and remove rules. The triggering event will not activate a deactivated rule. With the use of this function, users can choose which rules to deactivate when they're not needed. Reactivating the rule is possible with the activate command.

The rule is removed from the system using the drop command. Another way to allow for the activation, deactivation, or removal of an entire collection of rules is to organise them into named rule sets. Having a command that can initiate a rule or collection of rules through an explicit PROCESS RULES command that the user issues is also helpful.

The second question is whether the triggered action need to be carried out concurrently, in place of, ahead of, or in addition to the triggering event. When a trigger is employed in conjunction with an event that precedes it, it acts first. Applications such as detecting violations of constraints can make advantage of it. An after trigger can be used in applications like maintaining derived data and monitoring for particular events and conditions. It executes the trigger after the event has been completed. In applications like executing corresponding updates on base relations in response to an event that is an update of a view, a instead of trigger is utilised to execute the trigger rather than the event.

Whether the activity being carried out should be regarded as a separate transaction or as a component of the transaction that set off the rule is a related question. We'll make an effort to group the different choices. It is crucial to remember that not every option might be accessible for every active database system.

Assume for the moment that the triggering event takes place during the execution of a transaction. Prioritising the different possibilities for the relationship between the triggering event and the condition evaluation of the rule should come first. Since the action must be taken only after determining if the condition evaluates to true or false, the rule condition evaluation is also known as rule consideration. Three primary options are available for rule consideration:

1. Immediate consideration. The condition is assessed instantly and as a part of the same transaction as the triggering event. Three other categories apply to this case:
   - ❖ Prior to starting the triggering event, assess the situation.
   - ❖ After the triggering event has been executed, assess the situation.
   - ❖ As an alternative to carrying out the triggering event, evaluate the condition.
2. Deferred consideration. At the conclusion of the transaction including the triggering event, the condition is assessed. There can be a lot of triggered rules in this situation that are awaiting the evaluation of their conditions.
3. Detached consideration. The triggering transaction gives rise to a distinct transaction for the evaluation of the condition.

Notes

The following set of options concerns the relationship between evaluating the rule condition and executing the rule action. Again, there are three alternative outcomes in this case: detached, deferred, or immediate execution. The majority of systems in use select option 1. That is, the action is carried out right away if the condition is assessed and it returns true.

Another issue involving active database rules is the contrast between row-level rules and statement-level rules. One must decide whether the rule should be taken into account once for the entire statement or separately for each row (i.e., tuple) affected by the statement because SQL update statements, which function as triggering events, have the ability to define a group of tuples.

5.1.6 Spatial Database

Functionality that supports databases that track items in a multidimensional space is incorporated into spatial databases. Examples include the two-dimensional spatial descriptions of its objects, which range from countries and states to rivers, cities, roads, seas and other geographic features, in cartographic data-bases that store maps. Geographical Information Systems (GIS) are the systems that control geographic data and related applications; they are utilised in fields including environmental applications, transportation systems, emergency response systems and combat management.

Since temperatures and other meteorological data are tied to three-dimensional spatial positions, other databases, like meteorological databases for weather information, are three-dimensional. Objects that have spatial features that describe them and that have spatial relationships among them are often stored in a spatial database. The spatial links between the objects are significant, as they are frequently required while making database queries. Although, in general, a spatial database can refer to any n-dimensional space, we will restrict our discussion to two dimensions for simplicity.

A spatial database is designed to store and retrieve information on spatial objects, such as points, lines and polygons. Spatial data frequently includes satellite imagery. Spatial queries are those that are made on these spatial data and use spatial parameters as predicates for selection. A spatial query would ask, for instance, “What are the names of all bookstores within five miles of the College of Computing building at Georgia Tech?” While standard databases analyse numeric and character data, processing geographical data types requires additional capabilities.

An external geographic database that maps the company headquarters and each customer to a 2-D map based on their address may need to be consulted in order to process spatial data types that are typically outside the scope of relational algebra in order to process a query like “List all the customers located within twenty miles of company headquarters.”

Each consumer will actually be connected to a latitude and longitude position. Since conventional indexes are unable to organise multidimensional coordinate data, this query cannot be processed using a conventional B+-tree index based on customer zip codes or other nonspatial properties. As a result, databases specifically designed to handle spatial data and spatial queries are required.

The common analytical steps used to process geographic or spatial data are shown in the table below.

Notes

Analysis Type	Type of Operations and Measurements
Measurements	Distance, perimeter, shape, adjacency, and direction
Spatial analysis/statistics	Pattern, autocorrelation, and indexes of similarity and topology using spatial and nonspatial data
Flow analysis	Connectivity and shortest path
Location analysis	Analysis of points and lines within a polygon
Terrain analysis	Slope/aspect, catchment area, drainage network
Search	Thematic search, search by region

Measurement operations are used to determine the area, relative size of an object's components, compactness, or symmetry of single objects as well as the relative position of other objects in terms of distance and direction. To find spatial correlations within and among mapped data layers, spatial analysis operations—often utilising statistical methods—are performed. A prediction map, for instance, may be made to show where clients for specific goods are most likely to be found based on past sales and demographic data.

The shortest path between two points and the connectivity between nodes or regions in a graph can both be found with the use of flow analysis methods. Finding out whether a given set of points and lines falls into a given polygon is the goal of location analysis (location). The procedure entails creating a buffer around current geographic features, after which features are identified or chosen according to whether they are inside or outside the buffer's boundary. The topography of a particular location can be represented with an x, y and z data model known as a Digital Terrain (or Elevation) Model (DTM/DEM) using digital terrain analysis, which is used to create three-dimensional models.

A DTM's x, y and z dimensions stand in for the horizontal plane and the corresponding x, y coordinates' spot heights, respectively. These models can be applied to the analysis of environmental data or to the planning of engineering projects that need for knowledge of the topography. A user can look for objects inside a certain spatial area using spatial search. Thematic search, for instance, enables us to look for items associated with a specific subject or class, such as "Find all water bodies within 25 miles of Atlanta" when the category for the search is water.

Among spatial objects, there are also topological relationships. To choose items based on their spatial relationships, these are frequently employed in Boolean predicates. For instance, a condition like "Find all freeways that travel through Arlington, Texas" would require an intersects operation to identify which freeways (lines) intersect the city limit if a city boundary is represented as a polygon and freeways are represented as multilines (polygon).

Spatial Data Types and Models

The common data types and models for storing spatial data are briefly described in this section. There are three main types of spatial data. Due to their widespread use in commercial systems, these forms have become the de facto industry standard.

Map Data covers different geographic or spatial characteristics of map objects, such as an object's shape and its location. Points, lines and polygons are the three fundamental types of features (or areas). In the scale of a specific application, points are used to describe the spatial characteristics of objects whose locations correspond to a single pair of 2-d coordinates (x, y, or longitude/latitude). Buildings, cell towers and stationary vehicles are some instances of point objects, depending on the scale.

Notes

A series of point locations that change over time can be used to depict moving cars and other moving things. A series of connected lines can approximate the spatial features of things with length, such as highways or rivers. When representing the spatial properties of an item with a boundary, such as a nation, a state, a lake, or a city, a polygon is utilised. Keep in mind that depending on the level of detail, some objects, such cities or buildings, might be depicted as either points or polygons.

The descriptive information that GIS systems attach to map features is known as attribute data. Imagine, for instance, that a map has features that represent the counties of a US state (such as Texas or Oregon). Each county feature (object) may have the following attributes: population, major city or town, square miles, etc. States, cities, congressional districts, census divisions and other aspects of the map could all have additional attribute data.

Images made by cameras, such as satellite images and aerial photos, are examples of image data. These photos can have interesting objects, such highways and buildings, layered on them. Map features can also have images as properties. Other map features can have photos added to them so that when a user clicks on the feature, the image is displayed. Raster data frequently takes the form of aerial and satellite photos.

There are two main categories into which spatial information models can be categorised: field and object. Depending on the needs and the customary model selection for the application, either a field-based model or an object-based model is used to model a spatial application (such as remote sensing or highway traffic control). While object models have historically been used for applications such as transportation networks, land parcels, buildings and other objects that possess both spatial and non-spatial attributes, field models are frequently used to model continuous spatial data, such as terrain elevation, temperature data and soil variation characteristics.

Spatial Operators

When performing spatial analysis, spatial operators are used to record all the pertinent geometric details of the objects that are physically embedded in the space as well as the relationships between them. Operators are divided into three major groups.

Topological operators: When topological transformations are used, topological features remain unchanged. After transformations like rotation, translation, or scaling, their attributes remain unchanged. The base level of topological operators allows users to check for intricate topological relationships between regions with a large border, while the higher levels provide more abstract operators that let users query ambiguous spatial data without relying on the underlying geometric data model. Examples include open (region), close (region) and inside (point, loop).

Projective operators: To express predicates regarding the concavity/convexity of objects as well as other spatial relations, projective operators such as convex hull are utilised (for example, being inside the concavity of a given object).

Metric operators: Metric operators give a more detailed account of the geometry of the object. In addition to measuring the relative positions of various objects in terms of distance and direction, they are also used to measure several global features of single objects (such as the area, relative size of an object's sections, compactness and symmetry). Examples include length (arc) and distance (point, point).

Dynamic Spatial Operators: The operations carried out by the aforementioned operators are static in the sense that their application has no impact on the operands. For instance, the curve itself is unaffected by the length of the curve. The objects on which

Notes

dynamic operations act are changed. Create, destroy and update are the three basic dynamic operations. Updating a spatial object that can be divided into translate (shift position), rotate (change orientation), scale up or down, reflect (produce a mirror image) and shear (deform).

Spatial Queries: Requests for spatial information made using spatial operations are known as spatial queries. Three common forms of spatial searches are illustrated by the following categories:

Range query: identifies the objects of a specific type that are present within a specified spatial region or at a specified distance from a specified place. (For instance, look for all hospitals in the Metropolitan Atlanta region or all ambulances five miles away from an accident.)

Nearest neighbor query: locates the closest instance of a specific type of object at a specified location. (For instance, locate the police vehicle that is most convenient to the scene of the crime.)

Spatial joins or overlays: usually connects objects of two different types depending on some spatial condition, such as the objects' spatial intersection, overlap, or proximity. (For instance, find all houses that are two miles or less from a lake, or all townships on a main highway connecting two cities.)

Spatial Data Indexing

In order to make it simple to locate things in a specific spatial area, objects are arranged into a series of buckets (which correspond to pages of secondary memory) using a spatial index. A bucket region, or area of space holding all the objects kept in a bucket, exists for each bucket. The bucket regions, which are often rectangles, divide the space so that each point belongs to exactly one bucket for point data structures. A spatial index can be provided in essentially two different ways.

The database system has specialised indexing structures that enable effective search for data objects based on spatial search operations. These indexing structures would serve a similar purpose to typical database systems' B+-tree indexes. Grid files and R-trees are two examples of these indexing structures. Spatial join processes can be sped up by using specialised spatial indexes, also known as spatial join indexes.

The two-dimensional (2-d) spatial data is converted to single-dimensional (1-d) data so that conventional indexing techniques (B+-tree) can be employed, as opposed to developing brand-new indexing structures. Space filling curves are the techniques for transforming from 2-d to 1-d. We won't go into great depth about these techniques (see the Selected Bibliography for further references).

Next, we provide an overview of a few spatial indexing methods.

Grid Files. For indexing spatial data in two dimensions and greater n dimensions, gird files are utilised. The fixed-grid approach creates equal-sized buckets out of an n-dimensional hyperspace. The fixed grid is implemented using an n-dimensional array as the data structure. To handle overflows, the items whose spatial locations are entirely or partially within a cell can be stored in a dynamic structure. For data that is evenly dispersed, like satellite imagery, this structure is helpful. The fixed-grid structure, however, is rigid and its directory may be both sparse and extensive.

R-Trees. An extension of the B+-tree for k-dimensions, where k > 1, the R-tree is a height-balanced tree. The minimum bounding rectangle (MBR), which is the smallest rectangle with sides parallel to the coordinate system's (x and y) axis, is used to represent

Notes

spatial objects in the R-tree for two-dimensional (2-d) space. The following characteristics of R-trees are similar to those of B+-trees but are tailored to 2-dimensional spatial objects.

Each index record (or entry) in a leaf node has the structure (I, object-identifier), where I is the MBR for the spatial object whose identifier is object-identifier.

Except for the root node, every node must be at least halfway full. Since M/2 <= m <= M, a leaf node that is not the root should have m entries (I, object-identifier). Similarly, a non-leaf node that is not the root must have m entries (I, child-pointer), where M/2 <= m <= M and I is the MBR that has the union of all the rectangles in the node that child-pointer is pointing at.

The root node should contain at least two pointers unless it is a leaf node and all leaf nodes are at the same level.

Each and every MBR has a side that is perpendicular to the axes of the world coordinate system.

The quadtree and its derivatives are among other spatial storage structures. For the purpose of identifying the locations of various items, quadtrees often divide each space or subspace into sections of equal size.

Spatial Join Index. An index structure for a spatial join precomputes a spatial join operation and stores the pointers to the related objects. The performance of recurrent join queries over tables with slow update rates is enhanced by join indexes. To respond to requests like “Create a list of highway-river combinations that intersect,” spatial join conditions are utilised. These item pairs that satisfy the cross spatial relationship are found and retrieved using the spatial join. The results of spatial relationships can be computed once and stored in a table with the pairs of object identifiers (or tuple ids) that satisfy the spatial relationship, which is essentially the join index. This is because computing the results of spatial relationships typically takes a lot of time.

A bipartite graph G = (V1,V2,E) can be used to define a join index, where V1 holds the tuple ids from relation R and V2 contains the tuple ids from relation S. If a tuple corresponding to (vr,vs) in the join index exists, the edge set contains an edge (vr,vs) for vr in R and vs in S. All of the associated tuples are represented as connected vertices in the bipartite networks. Operations (see Section 26.3.3) that involve computing associations among spatial objects employ spatial join indexes.

Spatial Data Mining

Spatial data frequently exhibits strong correlation. For instance, individuals with comparable traits, professions and origins frequently congregate in the same areas.

Spatial classification, spatial association and spatial grouping are the three main spatial data mining approaches.

Spatial classification. Estimating the value of a characteristic of a relation based on the value of the relation’s other attributes is the aim of categorisation. Identifying the locations of nests in a wetland based on the importance of other qualities (such as vegetation durability and water depth) is an example of the spatial classification problem, also known as the location prediction problem. Similar to location prediction, predicting hotspots for criminal activity is a challenge.

Spatial association. In contrast to items, spatial predicates are used to establish spatial association rules. If at least one of the Pis or Qjs is a spatial predicate, then the rule is said to be of the type P1 P2 ^... ^Pn Q1^ Q2^ ... ^Qm. A good example of an association rule with some support and confidence is the rule is_a(x, country) ^ touches(x, Mediterranean)

Notes

is_a (x, wine-exporter) (that is, a country that is adjacent to the Mediterranean Sea is typically a wine exporter) is an example of an association rule, which will have a certain support s and confidence c.

In an effort to point to collecting data sets that are spatially indexed, spatial colocation rules try to generalise association rules. Between spatial and nonspatial linkages, there are a number of significant distinctions, including:

Given that data is embedded in continuous space, the concept of a transaction is absent in spatial contexts. Partitioning space into transactions would cause interest metrics, such support or confidence, to be overestimated or underestimated.

The size of item sets in spatial databases is minimal, meaning that in a spatial scenario there are significantly less items in the item set than in a non-spatial situation.

Spatial elements are often the discrete form of continuous variables. For instance, in the United States, income regions can be categorised as areas where the mean annual income falls within a given range, like $40,000 or less, $40,000 to $100,000, or $100,000 or more.

The goal of spatial clustering is to organise database objects into clusters where the most similar things are located and clusters where the least similar objects are located. The grouping of seismic occurrences in order to identify earthquake faults is one use for spatial clustering. Density-based clustering, which seeks to identify clusters based on the density of data points in a region, is an illustration of a spatial clustering technique. In the data space, these algorithms treat clusters as dense regions of items.

Density-based spatial clustering of applications with noise (DBSCAN) and density-based clustering are two variants of these methods (DENCLUE). Because it starts by identifying clusters based on the estimated density distribution of related nodes, DBSCAN is a density-based clustering algorithm.

Applications of Spatial Data

Numerous fields, including geography, remote sensing, urban planning and natural resource management, can benefit from spatial data management. The solving of complex scientific issues like global climate change and genomics is greatly aided by spatial database management. GIS and spatial database management systems have a significant role to play in the field of bioinformatics because of the geographical character of genetic data.

Pattern recognition, the building of genome browsers and map visualisation are some of the main applications. For instance, one might check to see if the topology of a specific gene in the genome is present in any other sequence feature map in the database. The detection of spatial outliers is one of the key applications of spatial data mining. A spatially referred object is considered an outlier if its nonspatial attribute values considerably deviate from those of other spatially referenced objects in its immediate vicinity. According to the nonspatial feature “house age,” for instance, if a neighbourhood of older homes has just one brand-new home, that home would be an outlier.

Many applications of geographic information systems and spatial data-bases make use of the ability to detect spatial outliers. Transportation, ecology, public safety, public health, climatology and location-based services are only a few of these application domains.

5.1.7 Deductive Database

Typically, a declarative language—one in which we define goals rather than methods of accomplishing them—is used to specify rules in deductive database systems. By analysing

Notes

these rules, an inference engine (also known as a deduction mechanism) built into the system can infer new facts from the database. The relational data model and in particular the domain relational calculus formalism, are strongly associated with the paradigm used for deductive databases.

It has connections to both the Prologue language and the discipline of logic programming. Prologue has been the foundation for logic-based deductive database work. Rules are defined declaratively in conjunction with an existing set of relations using Datalog, a Prologue variant where the relations are treated as literals in the language. While Datalog's language structure is similar to Prolog's, its operational semantics—that is, the way a Datalog programme is carried out—are distinct.

Rules and facts are the two primary categories of specifications used in a deductive database. Similar to how relations are stated, facts are specified with the exception that attribute names are not required. Remember that each tuple in a relation represents a fact about the real world whose meaning is influenced by the attribute names. In a deductive database, the meaning of an attribute value in a tuple is decided only by its position within the tuple.

Relational views and rules are comparable in certain ways. They define virtual relations that can be constructed from the facts by using inference processes based on the rule specifications; these relations are not really stored. The primary distinction between views and rules is that views cannot be stated in terms of fundamental relational views, whereas rules may entail recursion and so produce virtual relations.

Backward chaining is a strategy used to evaluate Prologue programmes that entails a top-down goal evaluation. Large quantities of data saved in a relational database have been handled with care in the deductive databases that employ Datalog. As a result, methods for evaluating have been developed that are similar to those for bottom-up evaluations. Prologue has the drawback that the order in which facts and rules are specified matters while evaluating them; also, the order in which literals within a rule matter. The methods used for Datalog programme execution aim to avoid these issues.

Prolog/Datalog Notation

The foundation of Prolog/Datalog notation is giving predicates distinct names. A predicate has a set number of parameters and an implicit meaning indicated by the predicate name. The predicate only asserts the truth of a particular fact if all of the arguments are constant values. In contrast, the predicate is regarded as a query or as a component of a rule or constraint if it takes variables as arguments. We follow the Prologue convention in our discussion, which states that all constant values in a predicate are either character strings or numbers; they are denoted by identifiers, or names, that begin with a lowercase letter, while variable names always begin with an uppercase letter.

Take a look at the illustration in the figure below. The predicate names are subordinate, oversee and superior. A series of facts is used to create the SUPERVISE predicate. Each fact has two arguments: the name of the supervisor and the name of the supervisor's direct supervisee (subordinate). These facts can be thought of as making up a set of tuples in a relation SUPERVISE with two attributes whose schema is: These facts match the real data that is kept in the database.

SUPERVISE(Supervisor, Supervisee)

(a)

Notes

Facts

SUPERVISE(franklin, john).
SUPERVISE(franklin, ramesh).
SUPERVISE(franklin, joyce).
SUPERVISE(jennifer, alicia).
SUPERVISE(jennifer, ahmad).
SUPERVISE(james, franklin).
SUPERVISE(james, jennifer).
. . .

Rules

SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Y*).
SUPERIOR(*X*, *Y*) :– SUPERVISE(*X*, *Z*), SUPERIOR(*Z*, *Y*).
SUBORDINATE(*X*, *Y*) :– SUPERIOR(*Y*, *X*).

Queries

SUPERIOR(james, *Y*)?
SUPERIOR(james, joyce)?

(b)

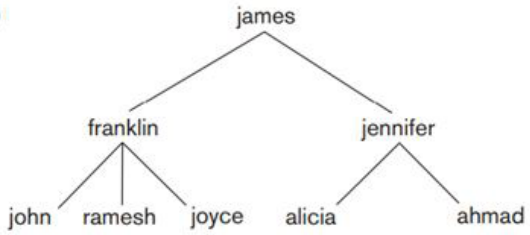


Figure: (a) Prolog notation. (b) The supervisory tree.

The notation SUPERVISE(X, Y) indicates that X is in charge of Y. Notice the lack of the attribute names in the Prologue notation. Only the order in which each argument appears in a predicate determines the representation of an attribute name: the supervisor is represented by the first argument and the direct subordinate is represented by the second.

Rules define the other two predicate names. The capacity to define recursive rules and the provision of a framework for deriving new data from the stated rules are the primary contributions of deductive databases. A rule has the format head:-body, where the prefix "if" (read as "only if") is used. A rule normally has a single predicate to the left of the :– symbol—called the head or left-hand side (LHS) or conclusion of the rule—and one or more predicates to the right of the :– symbol— termed the body or right-hand side (RHS) or premise(s) of the rule.

Ground, also known as instantiated predicate, is a predicate that takes constants as arguments. Although predicates can also have constants as arguments, the arguments of the predicates that appear in a rule usually consist of several variable symbols. According to a rule, if a certain assignment or binding of constant values to the variables in the body (RHS predicates) makes all of the RHS predicates true, then applying the same assignment of constant values to variables also makes the head (LHS predicate) true.

Therefore, a rule gives us a mechanism to produce new facts that are instances of the rule's head. The instantiations (or bindings) of predicates in the body of the rule correspond to the facts upon which these new facts are predicated. Observe that we implicitly use the logical AND operator to the predicates when we list multiple predicates in the body of a rule. As a result, one could interpret the commas as meaning and between the RHS predicates.

Examine the definition of the predicate SUPERIOR in the above Figure. Its first argument is the name of an employee and its second argument is an employee who is the first employee's direct or indirect subordinate. The subordinate of another subordinate down to any number of levels is referred to as an indirect subordinate. Hence, SUPERIOR(X, Y) denotes the circumstance in which X oversees Y directly or indirectly.

Two rules that together define the meaning of the new predicate can be written. The first rule under Rules in the image indicates that, since Y would be X's immediate subordinate (at one level below), if SUPERVISE(X, Y)—the rule body—is true for every value of X and Y, then SUPERIOR(X, Y)—the rule head—is likewise true. The facts that constitute the SUPERVISE predicate can be utilised to establish all direct superior/

Notes

subordinate connections. According to the second recursive rule, SUPERIOR(X, Y) is also true if SUPERVISE(X, Z) and SUPERIOR(Z, Y) are both true.

One of the rule head predicates in the LHS and one of the rule body predicates in the RHS of this recursive rule are the same. Generally speaking, the rule head's conclusion can be inferred from the rule body's definition of several premises if all of them are true. Notice that if we have two (or more) rules with the same head (LHS predicate), it is equal to declaring that the predicate is true (that is, that it may be instantiated) if either one of the bodies is true; consequently, it is equivalent to a logical OR operation.

As an illustration, two rules X: Y and X: Z are identical to one rule X: Y OR Z. The latter form is not utilised in deductive systems, however, because it is not in the usual form of rule, termed a Horn clause.

Numerous predicates that are automatically understood by a Prologue system are present. Typical examples of these are the equality comparison operator = (X, Y), which can be expressed as X = Y using conventional infix notation and returns true if X and Y are identical. Binary predicates may be applied to other comparison operators for numbers, such as <=, > and >=. In Prologue predicates, arithmetic functions like +, –, * and / can be used as inputs. The inability of Datalog (in its most basic form) to accept functions like arithmetic operations as arguments is one of the key distinctions between Datalog and Prologue. Nonetheless, there have been suggestions for Datalog extensions that do contain functions.

The meaning (or answer) of a query usually consists of a predicate symbol and some variable arguments and it deduces all the possible constant combinations that, when bound (assigned) to the variables, can make the predicate true. For instance, the first question in the above Figure asks for the names of all of James's subordinates, regardless of rank. An alternative kind of inquiry, with just constant symbols as parameters, can provide a true or false answer, based on whether or not the arguments can be inferred from the available facts and rules. For instance, the second query in the previous Figure yields a true result since it is possible to deduce SUPERIOR(james, Joyce).

Datalog Notation

Similar to other logic-based languages, Datalog also uses atomic formulae as the building blocks from which programmes are constructed. Typically, atomic formula syntax is defined by showing how they can be joined to construct programmes and this is how logic-based language syntax is defined. Atomic formulae in Datalog are literals of the format p(a1, a2,..., an), where n is the number of arguments for predicate p and p is its name. The number of arguments (n) of predicate p is sometimes referred to as the arity or degree of p. Various predicate symbols can have varying numbers of arguments. Variable names or constant values may be used as arguments. As previously stated, we follow the practice that variable names always begin with an uppercase character, but constant values must either be numeric or begin with a lowercase character.

Datalog comes with several predicates that are built-in and can be utilised to create atomic formulas. There are two primary categories of built-in predicates: the comparison predicates = (equal) and /= (not_equal) over ordered or unordered domains and the binary comparison predicates < (less), \= (less_or_equal), > (greater) and >= (greater_or_equal) over ordered domains. These can be given using the standard infix notation X<3, or they can be used as binary predicates with the same functional syntax as other predicates, for example, by writing less(X, 3).

Notes

It should be noted that these predicates should be utilised cautiously in rule definitions because their domains may be endless. For instance, when the predicate greater(X, 3) is applied alone, it yields an infinite set of values for X (any integer numbers greater than 3) that fulfil the predicate.

A literal might be an atomic formula preceded by not or an atomic formula as stated above (referred to as a positive literal). The latter is referred to as a negative literal and is a negated atomic formula. A subset of the predicate calculus formulae, which resemble the domain relational calculus formulas in certain ways, can be thought of as datalog programmes. However, before being represented in Datalog, these formulas must first be transformed into a form known as clausal form. Only formulas provided in a restricted clausal form, known as Horn clauses, are allowed to be utilised in Datalog.

5.1.8 Multimedia Database

Multimedia databases offer features that enable users to store and query various types of multimedia information, including documents, audio clips, video clips and home videos. Finding multimedia sources that include specific objects of interest is the primary use for database queries. For instance, it could be necessary to find every video clip in a video library featuring, let's say, Michael Jackson. It could be desirable to obtain video clips that feature specific actions, like recordings in which a particular player or team scores a goal in football.

Since specific objects or activities are being retrieved from the multimedia source based on its presence, these types of queries are known as content-based retrieval. As a result, a multimedia database has to index and arrange the multimedia sources according to some paradigm. Multimedia source content identification is a challenging and time-consuming task. There are two primary methods.

The first relies on automatically analysing multimedia sources to pinpoint specific mathematical features present in their content. Depending on the kind of multimedia source (image, video, audio, or text), this method employs several strategies.

The second method relies on manually identifying the noteworthy items and activities in every multimedia source, then indexing the sources using this data. This method works with any multimedia source; however, it necessitates a manual preprocessing step where each source must be scanned to find and classify the objects and activities it contains. From there, the sources can be indexed.

Usually, an image is saved in compressed form to conserve space, or in raw form as a collection of pixel or cell values. The raw image's geometric shape, which is usually a rectangle made up of cells with a specific width and height, is described by the image shape descriptor. Thus, a grid of cells measuring m by n can be used to represent any image. A pixel value that describes the content of each cell is present in every cell. Pixels in black-and-white photos can be one bit. A pixel in a colour or grayscale image is made up of many bits.

Images are frequently stored in compressed form since they may need a lot of space. Compression standards, like GIF, JPEG, or MPEG, use a variety of mathematical operations to decrease the amount of stored cells while preserving the primary features of the image. Discrete Fourier transform (DFT), discrete cosine transform (DCT) and wavelet transforms are among the applicable mathematical transformations.

An picture is usually segmented using a homogeneity predicate into homogeneous segments in order to identify objects of interest. For instance, neighbouring cells with comparable pixel values in a colour image are grouped together into a segment. The requirements for automatically classifying those cells are specified by the homogeneity

Notes

predicate. Therefore, segmentation and compression can be used to determine an image's primary features.

Finding photographs in the database that are comparable to a particular image is a common query in image databases. Finding more photographs with the same pattern could be the task given. For example, the given image could be a solitary fragment with an interesting pattern. There are two primary methods for conducting this kind of search. In the first method, the provided image is compared with the stored images and their segments using a distance function. There is a strong likelihood of a match if the distance value returned is minimal. To reduce the size of the search space, indexes can be made to classify saved images that are similar in terms of distance.

Using a limited set of transformations to match the cells in one image to those in the other, the second method, dubbed the transformation approach, calculates how similar two images are. Rotations, translations and scaling are examples of transformations. The transformation strategy is more complex and time-consuming, but it is also more general.

Usually, a video source is shown as a series of frames, with each frame being a still image. However, the movie is broken into video segments, each of which consists of a series of consecutive frames with the same items and activities, as opposed to listing the things and activities in each individual frame. The beginning and ending frames of each section serve as identifiers.

The portions of the video can be indexed using the items and actions noted in each part. For video indexing, a method known as frame segment trees has been suggested. The index covers both activities, like a person giving a speech or two people conversing, as well as things, such people, houses and cars. Another common compression method for videos is to use an MPEG standard.

Audio sources can be pre-recorded lectures, presentations in class, or even police surveillance recordings of phone calls or conversations. Here, the primary features of an individual's speech can be determined using discrete transforms to enable similarity-based indexing and retrieval.

In essence, a text/document source is an article, book, or magazine's entire text. The usual method for indexing these sources is to determine the relative frequencies of the terms that appear in the text. Stopwords, which are common words or filler terms, are removed from the process. In an effort to index a collection of documents, methods have been devised to narrow the number of keywords to those that are most pertinent to the collection because there can be a large number of keywords. Singular value decomposition (SVD), a dimensionality reduction method based on matrix transformations, can be applied in this way. Then, comparable documents can be grouped using an indexing method known as telescoping vector trees (TV-trees).

Automatic Analysis of Images

For any kind of query or search interface to function, multimedia source analysis is essential. Multimedia source material, such photos, must be represented using features that allow us to establish similarity. Low-level visual elements like colour, texture and shape are used in the work done in this area thus far and they are closely tied to the perceptual aspects of image content. It is simple to extract and represent these features and to create similarity measures based on their statistical characteristics.

Colour is one of the most often used visual features in content-based picture retrieval since it does not depend upon image size or orientation. The primary method for performing color-similarity-based retrieval is to compute a colour histogram for every image, which

Notes

shows the percentage of pixels in each of the three RGB colour channels (red, green and blue). Nonetheless, the orientation of the object with relation to illumination and camera direction has an impact on RGB representation. For this reason, competing invariant representations like HSV (hue, saturation, value) are used in contemporary picture retrieval approaches to compute colour histograms.

According to HSV, colours are represented as points inside a cylinder whose centre axis runs from neutral colours in the middle to black at the top and white at the bottom. Hue, saturation and brightness are correlated with the angle around the axis, distance from the axis and distance along the axis, respectively (brightness).

Patterns in a picture that exhibit homogeneity without coming from the existence of a single colour or intensity value are referred to as textures. Texture classes include things like smooth and rough. It is possible to identify textures such as pressed calf leather, straw matting, cotton canvas and so forth. Similar to how arrays of pixels (picture elements) represent pictures, arrays of texels (texture elements) represent textures.

Depending on how many textures are found in the image, these textures are then grouped into several groups. These sets provide the texture definition as well as the location of the texture within the image. The most common method for identifying textures is to model them as gray-level variations in two dimensions. The degree of contrast, regularity, coarseness and directionality are estimated by computing the relative brightness of pairs of pixels.

Shape describes a region's shape inside a picture. Usually, segmentation or edge detection are used to determine it on an image. While edge detection is a boundary-based technique that employs only the outer boundary properties of entities, segmentation is a region-based approach that uses a complete region (sets of pixels). Generally, shape representation needs to be scaling, rotation and translation invariant. Moment invariants and Fourier descriptors are two popular techniques for representing shapes.

Object Recognition in Images

Finding real-world things in an image or a video sequence is known as object recognition. Even if the object's images are rotated, translated, or have different views, sizes, or scales, the system still needs to be able to recognise it. Various methods have been devised to partition the initial image into areas according to the resemblance of adjacent pixels. As a result, in a given image of a tiger in the jungle, a subimage of the tiger may be identified against the jungle's background and, upon comparison with a collection of training images, it may be identified as such.

It is crucial that the multimedia item be represented in an object model. Using a homogeneous predicate, one method is to split the image into homogeneous segments. For instance, neighbouring cells with comparable pixel values in a coloured image are grouped together into a segment. The requirements for automatically classifying those cells are specified by the homogeneity predicate. Therefore, segmentation and compression can be used to determine an image's primary features.

An alternative method identifies measures of the object that remain constant across modifications. Maintaining a database with examples of every potential transformation for an image is not feasible. Object recognition techniques discover interesting points (or characteristics) in a picture that are not affected by changes in order to address this.

An notable addition to this subject was made by Lowe, who exploited scale-invariant characteristics from photos to conduct reliable object recognition. The scale-invariant feature transform (SIFT) is the name of this methodology. The SIFT features are partially

Notes

invariant to changes in illumination and 3D camera viewpoint and they are invariant to scaling and rotation of the image. They are less likely to be disturbed by occlusion, clutter, or noise because they are well-localised in both the spatial and frequency domains. A single characteristic may be accurately matched with a high probability against a huge database of features due to its great distinctiveness, which also serves as a foundation for object and scene identification.

First, a series of reference photos is used to extract SIFT features, sometimes called keypoint features, which are then saved in a database for image matching and recognition. After that, each feature from the fresh image is compared to the features kept in the database in order to identify potential matching features. This is done by calculating the Euclidean distance between each feature vector's feature vectors. A single feature can be accurately matched with a good probability in a huge database of features due to the great distinctiveness of the keypoint features.

Several competing approaches are available for object recognition under partial occlusion or clutter in addition to SIFT. For instance, groups of local affine regions (image features with a distinctive appearance and elliptical shape) that hold roughly affinely rigid across a variety of views of an object and across multiple instances of the same object class are identified by RIFT, a rotation invariant generalisation of SIFT.

Semantic Tagging of Images

Implicit tagging is a crucial concept for image comparison and recognition. An image or a subimage may have more than one tag attached to it. For example, in the previously mentioned example, the tags "tiger," "jungle," "green," and "stripes" may all be attached to the same image. The majority of image search algorithms fetch photos using user-supplied tags, which are frequently neither extremely precise nor thorough. A number of contemporary systems try to generate these picture tags automatically in order to increase search quality.

The majority of the semantics of multimedia data are found in its content. These systems analyse the content of images using statistical modelling and image processing techniques to provide precise annotation tags that may be used to retrieve images based on content. The quality of picture retrieval will be low since different annotation techniques will annotate photos using diverse vocabularies. The usage of concept hierarchies, taxonomies, or ontologies using OWL (Web Ontology Language), which defines terms and their relationships precisely, have been suggested as solutions to this issue by contemporary research methodologies.

Based on tags, they can be used to infer higher-level concepts. In such a taxonomy, terms like "sky" and "grass" could be further subdivided into "cloudy sky" and "clear sky," or "dry grass" and "green grass." These methods, which generally fall under semantic tagging, can be applied in addition to the feature-analysis and object-identification techniques mentioned above.

Analysis of Audio Data Sources

Speech, music and other audio data are the general categories into which audio sources fall. Since each of them differs greatly from the others, distinct categories of audio data are handled in different ways. Before audio data can be processed and stored, it needs to be digitalised. Audio data is likely the hardest to index and retrieve out of all the media kinds since, similar to film, it is continuous in time and lacks easily observable features like text.

Although it is simple for humans to detect, machine learning finds it difficult to measure sound recording clarity. It's interesting to note that speech recognition algorithms are

Notes

frequently used to support audio content in speech data because they can greatly improve the ease and accuracy of indexing this data. This is sometimes called text-based audio data indexing. Speech metadata is usually content based, meaning that the information about the speech's length, number of speakers and other details is derived from the audio content. Some metadata, such the speech's duration and the format in which the data is kept, may, nevertheless, be unrelated to the content itself.

Contrarily, music indexing—also referred to as content-based indexing—is carried out via statistical analysis of the audio stream. Pitch, timbre, rhythm, intensity and other essential aspects of sound are frequently used in content-based indexing. Different audio data sets can be compared in order to extract information through the application of specific transforms and the calculation of specific features.

Summary

- Multilevel transactions are a complex aspect of database management, especially in distributed and hierarchical database systems. Proper design, coordination and concurrency control mechanisms are essential to ensure the integrity of the database during multilevel transactions.
- Long-lived transactions, also known as sagas, are a design pattern employed in distributed systems to manage and coordinate a series of related transactions that span an extended period. In contrast to traditional atomic transactions, sagas break down the overall transaction into smaller, independent steps, each associated with a compensating transaction to undo its effects in case of failure.
- Data warehousing involves the collection, transformation and storage of data from diverse sources in a centralised repository, known as a data warehouse. This structured and optimised storage enables efficient analysis and reporting for decision-making within organisations.
- Data mining is the process of discovering meaningful patterns, trends and insights from large volumes of data through advanced statistical techniques, machine learning algorithms and computational methods. It involves extracting valuable knowledge from complex datasets to uncover hidden relationships and make predictions or informed decisions.
- An active database refers to a database system that not only stores and retrieves data but also incorporates a set of rules and procedures for automatically responding to events or changes in the database environment. Unlike traditional, passive databases, active databases actively monitor the database for predefined events, triggers, or conditions and execute predefined actions in real-time or near-real-time
- Spatial databases find applications in diverse fields, including geographic information systems (GIS), urban planning, environmental science, logistics and location-based services. They play a crucial role in storing, managing and analysing spatial data, enabling users to make informed decisions based on the geographic context of their information.
- A deductive database is a type of database management system that extends traditional relational databases with logical reasoning capabilities based on formal logic and rules. Unlike standard databases that primarily focus on data storage and retrieval, deductive databases incorporate a rule-based system that allows users to define relationships, dependencies and inferential logic.
- A multimedia database is a specialised database system designed to efficiently store, manage and retrieve multimedia data, which includes a combination of text, images, audio, video and other forms of non-traditional data types.

Notes

Glossary

- DBMS: Database Management System
- EOT: End-Of-Transaction
- ACID: Atomicity, Consistency, Isolation, Durability
- UDF: User-Defined Functions
- BLOB: Binary Large Objects
- IoT: Internet of Things
- DM: Data Mining
- MBA: Market Basket Analysis
- KNN: K-Nearest Neighbor
- ECA: Event-Condition-Action
- GIS: Geographical Information Systems
- MBR: Minimum Bounding Rectangle
- DFT: Discrete Fourier transform
- SVD: Singular value decomposition
- HSV: Hue, Saturation, Value

Check Your Understanding

1. In multilevel transactions, what does “multilevel” refer to?
   a) Multiple transaction participants
   b) Multiple database tables
   c) Multiple levels of security classification
   d) Multiple transaction phases
2. What is the primary challenge addressed by multilevel transactions?
   a) Efficient indexing
   b) Concurrency control
   c) Data normalisation
   d) Handling data at different security levels
3. Which of the following is a key feature of multilevel transactions?
   a) High-speed processing
   b) Dynamic schema changes
   c) Support for transactions with different security clearances
   d) Schema redundancy elimination
4. What is the main purpose of a Saga in the context of long-lived transactions?
   a) Ensuring data consistency
   b) Managing concurrent transactions
   c) Handling failures and compensating transactions
   d) Simplifying database queries
5. In the context of Sagas, what does the term “compensating transaction” refer to?
   a) A transaction that completes successfully

b) A transaction that undoes the effects of a preceding transaction

c) A transaction with multiple participants

d) A long-running query

Notes

Exercise

1. Define:

a. Multilevel Transactions

b. Long-lived Transactions (Saga)

2. What do you mean by data warehousing and data mining?

3. Define active database and spatial database.

4. Define deductive database and multimedia database.

Learning Activities

1. You are the data analyst on the project team building a data warehouse for an insurance company. List the possible data sources from which you will bring the data into your data warehouse. State your assumptions.

2. Imagine you are tasked with creating a deductive database system for a university's course enrollment. Students have certain prerequisites for enrolling in courses and the deductive database will infer whether a student is eligible based on these prerequisites.

Check Your Understanding- Answers

1. c 2. d 3. c 4. c

5. b