SANT DNYANESHWAR SHIKSHAN SANSTHA’S

ANNASAHEB DANGE COLLEGE OF ENGINEERING & TECHNOLOGY, ASHTA



## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# **LABORATORY MANUAL**

**Advanced Database SystemS**

## THIRD YEAR – I

**2019 – 2020**

**SEM II**

**EXPERIMENT NO. 1**

**TITLE: -** Implementation of Centralized Database Architecture.

**AIM** **: -** To study & Implement Centralized Database Architecture concept.

**THEORY:-**

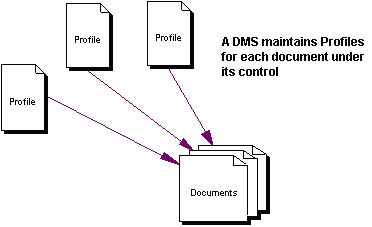
**The Role of the Database in Document Management Systems**

Document management software is designed coordinate and control the documents created, maintained, and used within a firm or organization. Virtually all electronic document management systems also offer additional functionality as a rule—including version control, document archiving, full-text indexing, content-based retrieval, network mirroring, workflow, and so forth. But the heart and soul of document management consists of cataloging and tracking documents.

This means that the document manager must somehow extract or derive "information about the documents," often referred to as *document profiles* or *metadata*, keeping it separate from the information in the documents themselves. It is critical to distinguish document **profile** information from document **content**. The critical role that documents serve within an organization cannot be over-stated. Documents are a firm's intellectual assets—in much the same way that staff are human assets. In order to manage employees effectively, firms maintain human resource records—frequently handled by an entire department dedicated to that function. A document management system performs an analogous function by maintaining *document* resource records.

The document manager must have the means to store, edit and retrieve the document resource records, or profile information, for the documents under its control. The obvious solution is to use a *database* system of some kind. Storing profile information in a database affords all the advantages inherent in database technology to the document manager. By means of a database the document manager can handle a vast amount of information that can be stored, organized,

and searched quickly and efficiently by large numbers of users. The database contributes an element of structure in parallel with the document repository, which consists of largely unstructured data.



In determining an approach to implementing database technology for a document management system (DMS), the essential questions must revolve around how best to leverage:

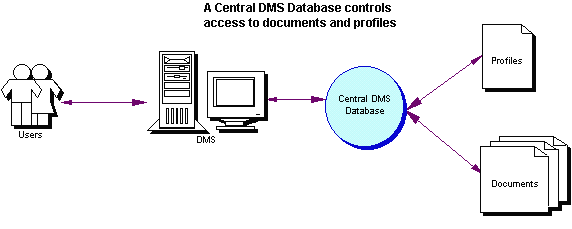
* the informational requirements inherent in managing a large document set
* available system resources at a DMS client site
* the skill level of information system staff (to support a particular database implementation)
* the day-to-day realities of document flow and usage experienced by users of the DMS solution.

Keep in mind that document management system databases or *not generally end-user facing*, meaning that most users do not have any direct experience with the database in typical usage scenarios. No more, say, than most automobile drivers have direct experience of their car's engine while driving. When it works as designed, the driver can disregard it and concentrate on getting from one place to another. As with automobile engines, the database engine of a DMS should be unobtrusive, efficient, powerful, and reliable. It should enable customers to achieve desired outcomes at an acceptable performance level without draining resources excessively or breaking down when needed most.

Central Database: Single Point of Access, Single Point of Failure

A document management system that employs a central database stores all document profile information in a single, monolithic database. Typically this is a relational database management system (RDBMS) that resides on a dedicated Server attached to the network. As users of the document management system work with documents across the network, the DMS routes all document profile updates and information requests to the server housing the central database.

Typically the database is an adjunct to the documents, which tend to remain distributed throughout the network. Database records therefore contain a pointer of some sort that "attaches" the record to the corresponding document's physical location. An extreme application of a centralized database approach to DMS, however, foregoes this level of abstraction and actually stores the documents within the database structure, often as a binary large object, or BLOB. While highly secure, this approach can also increase risk of lost documents or impeded productivity during a hardware or software breakdown.



A centralized database offers several key benefits:

* controlled access to the document profile repository
* quick, efficient searching

Many DMS vendors have constructed systems that use SQL database technology. Ostensibly, this is to enhance interoperability and to take advantage of the client organization's existing expertise with the technology, if present. SQL databases offer these benefits:

* some sites can leverage existing SQL proficiency (if present)
* scaleable support for Wide Area Networks
* offers opportunity to consolidate data stores.

On the down side, a single, centralized database containing the entire body of information about a firm's document repository presents a single point of failure. Without a properly defined and strictly enforced backup regime, the failure or corruption-for any reason-of a centrally stored document profile repository, can be catastrophic. If the sole document profile database is corrupted, restoring any lost information becomes a drop-everything, mission-critical operation. Even with current backups, at a minimum the DMS-and probably the network as well-will have to be "brought down," bringing productivity to a standstill. While the potential for a "doomsday" DMS scenario is not necessarily high in well implemented and well maintained systems, it *does* happen from time to time. Stories abound of firms having to stop all work, search for the latest backup tapes-or worse, of having to recreate document profiles from scratch. The more likely scenario on the downside is that the server hosting the centralized DBMS may go down. Without a redundant system in place, such as mirroring, the loss of the server will generally enforce a network-wide work stoppage.

In large organizations with the staff and the know-how, an SQL back-end to a DMS can prove beneficial on several fronts. However, in smaller organizations, or those lacking in-house SQL expertise, a dedicated SQL database can prove to be a serious resource drain, both on the network and on budgets that are bound to expand in order to accommodate outsourcing SQL database expertise. SQL databases are powerful, but that power comes at a cost.

Be assured that installing an SQL database is in no way a turnkey operation. It takes studied expertise to set up the database, and ongoing tuning, backups, and continual upkeep in order to keep it running within acceptable performance limits.

It is the presence of these all too real concerns that has led to another approach…

**PROGRAM :**

import java.sql.\*;

class OracleCon{

public static void main(String args[]){

try{

//step1 load the driver class

Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 create the connection object

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

//step3 create the statement object

Statement stmt=con.createStatement();

//step4 execute query

ResultSet rs=stmt.executeQuery("select \* from emp");

while(rs.next())

System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));

//step5 close the connection object

con.close();

}catch(Exception e){ System.out.println(e);}

}

}

**CONCLUSION:-** The Centralized DBMS & architectures provides use of databases & query evaluation in more reliable way.

**EXPERIMENT NO. 2**

**TITLE: -** Create structured data types in ORDBMS

**AIM** **:** To study & Implement create table using Structured data types, insert data and solve queries.

**Problem Statement:**

Create and Query Object Relational Tables

EMP (Employee Table)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| COL NAME | TYPE | SIZE | NULL | DESCRIPTION |
| EMPNO | CHAR | 6 | NO | Employee Number, Unique |
| FIRSTNAME | VARCHAR | 15 | NO | First Name |
| LASTNAME | VARCHAR | 15 | NO | Last Name |
| WORKDEPT | CHAR | 3 |  | Employees dept number |
| SEX | CHAR | 1 |  | M-Male, F-Female |
| BIRTHDATE | DATE |  |  | Date of birth |
| SALARY | NUMBER | (8, 2) |  | Annual Salary |

DEPT (Department Table)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| COL NAME | TYPE | SIZE | NULL | DESCRIPTION |
| DEPTNO | CHAR | 3 | NO | Department Number, Unique |
| DEPTNAME | VARCHAR | 30 | NO | Department Name |
| MGRNO | CHAR | 6 |  | Dept managers employee no |
| ADMRDEPT | CHAR | 3 |  | ID of administrator dept |

1. Define Object type emp\_type and dept\_type with attributes of EMP and DEPT respectively.
2. Create table named employee\_table and department\_table using Object type defined in (a).
3. Insert data into the Object relational tables created in (b).

Answer the following queries using EMP and DEPT tables:

a) Get the department name and manages last name for all department.

b) Get the employee number, last name and the department name of every employee.

c) For each department , display the department number , department name and name of the administrative department.

d) For each department, display the department number, department name and name of the administrative department and the last name of the manager of the administrative department.

e) Display employee number , firstname, lastname and salary of every employee along with lastname and salary of the manager of the employees work department.

f) Show the average salary for men and the average salary for women for each department. Identify the department by both department number and name.

=======CREATING TYPE=======

CREATE TYPE dept\_Type

/

CREATE TYPE employee\_Type AS OBJECT(

empno char(6),

firstname varchar(12),

lastname varchar(15),

workDept REF dept\_Type,

sex char(1),

birthDate date,

salary number(8,2)

)

/

CREATE TYPE dept\_Type AS OBJECT(

deptNo char(3),

deptName varchar(36),

mgrNo REF employee\_Type,

admrDept REF dept\_Type

)

/

SQL> desc dept\_Type // It will describe the type

=================================================

=======Creating Tables and Constraints======

CREATE TABLE EMPLOYEE\_TABLE OF employee\_Type(

CONSTRAINT tblemp\_primarykey PRIMARY KEY ( empno ),

CONSTRAINT tblemp\_fname firstname NOT NULL,

CONSTRAINT tblemp\_lname lastname NOT NULL,

CONSTRAINT tblemp\_check\_sex CHECK ( sex = 'M' OR sex = 'm' OR sex = 'F' OR sex = 'f')

)

/

SQL> desc employee\_table // It describes the table

CREATE TABLE DEPARTMENT\_TABLE OF dept\_Type(

CONSTRAINT tbldept\_prinarykey PRIMARY KEY (deptno),

CONSTRAINT tbldept\_mgrno\_fk FOREIGN KEY (mgrNo) REFERENCES EMPLOYEE\_TABLE,

CONSTRAINT tbldept\_admin\_fk FOREIGN KEY (admrDept) REFERENCES DEPARTMENT\_TABLE,

CONSTRAINT tbldept\_deptno deptNo NOT NULL,

CONSTRAINT tbldept\_deptname deptName NOT NULL

)

/

SQL> desc department\_table

ALTER TABLE EMPLOYEE\_TABLE ADD CONSTRAINT tblemp\_workdept FOREIGN KEY (workDept)REFERENCES department\_table

/

=====================================================

============= Inserting Types into tables============

INSERT INTO DEPARTMENT\_TABLE VALUES (dept\_Type('A01','CSE',NULL,NULL))

/

INSERT INTO DEPARTMENT\_TABLE VALUES (dept\_Type('A02','ETC',NULL,(SELECT REF (d) FROM DEPARTMENT\_TABLE D WHERE d.deptno = 'A01')))

/

INSERT INTO DEPARTMENT\_TABLE VALUES (dept\_Type('A03','CIVIL',NULL,(SELECT REF (d) FROM DEPARTMENT\_TABLE D WHERE d.deptno = 'A01')))

/

INSERT INTO DEPARTMENT\_TABLE VALUES (dept\_Type('A04','MECH',NULL,(SELECT REF (d) FROM DEPARTMENT\_TABLE D WHERE d.deptno = 'A01')))

/

======================================================

===============Handling Complex Inserts and Updates===

UPDATE DEPARTMENT\_TABLE d

SET d.admrDept = (

SELECT REF(d) FROM DEPARTMENT\_TABLE d

WHERE d.deptNo = 'A01'

) WHERE d.deptNo = 'A01'

/

INSERT INTO EMPLOYEE\_TABLE values (employee\_Type('110011','Sachin','Kulkarni',(SELECT

REF (d) FROM DEPARTMENT\_TABLE WHERE d.deptNo = 'A01'),'M','15-AUG-85','500000'))

/

**T1 Batch: Problem Statement:**

A car-rental company maintains a database for all vehicles in its current fleet. For all vehicles, it includes the vehicle identification number, license number, manufacturer, model, date of purchase, and color. Special data are included for certain types of vehicles:

• Trucks: cargo capacity.

• Sports cars: horsepower, renter age requirement.

• Vans: number of passengers.

• Off-road vehicles: ground clearance, drivetrain (four- or two-wheel drive).

Construct an SQL schema definition for this database. Use inheritance where appropriate.

1. Define Object type.
2. Create table using Object type defined in (a).
3. Insert data into the Object relational tables created in (b).

**T2 Batch: Problem Statement:**

Consider a database schema with a relation *Emp* whose attributes are as shown below, with types specified for multivalued attributes.

*Emp = (ename, ChildrenSet* **multiset***(Children), SkillSet* **multiset***(Skills))*

*Children = (name, birthday)*

*Skills = (type, ExamSet* **setof***(Exams))*

*Exams = (year, city)*

a. Define the above schema in SQL, with appropriate types for each attribute.

b. Using the above schema, write the following queries in SQL.

i. Find the names of all employees who have a child born on or

after January 1, 2000.

ii. Find those employees who took an examination for the skill type

“typing” in the city “Dayton”.

iii. List all skill types in the relation *Emp*.

**T3 Batch: Problem Statement:**

Consider the relational schema shown in below

a. Give a schema definition in SQL corresponding to the relational schema, but using references to express foreign-key relationships.

*employee* (*person name*, *street*, *city*)

*works* (*person name*, *company name*, *salary*)

*company* (*company name*, *city*)

*manages* (*person name*, *manager name*)

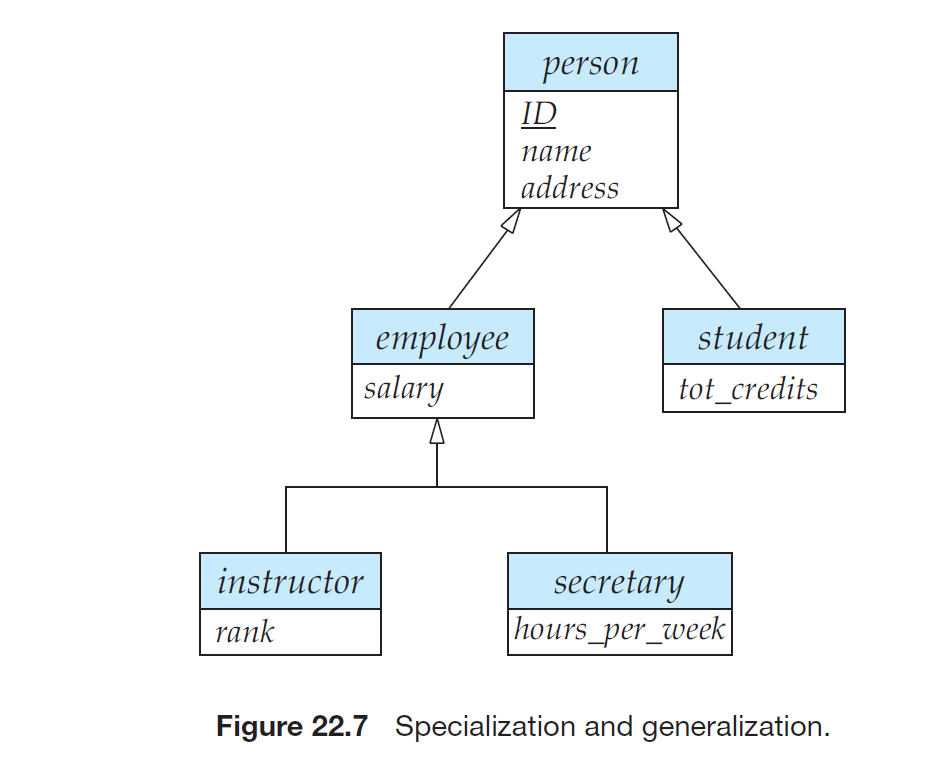
b. Write each of the queries given in below on the above schema, using SQL.

a. Find the company with the most employees.

b. Find the company with the smallest payroll.

c. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

**T4 Batch: Problem Statement:**



Consider the E-R diagram in Figure 22.7, which contains specializations, using subtypes and sub-tables.

a. Give an SQL schema definition of the E-R diagram.

b. Give an SQL query to find the names of all people who are not secretaries.

c. Give an SQL query to print the names of people who are neither employees nor students.

d. Can you create a person who is an employee and a student with the schema you created? Explain

How, or explain why it is not possible.

**EXPERIMENT NO. 3**

**TITLE: -** Make use of multivalued attributes, complex types and Inheritance in ORDBMS.

**AIM** **:** To study & Implement multivalued attributes, complex types and Inheritance in ORDBMS.

**Theory:**

**ORDBMS Definition:**

An object relational database is also called an object relational database management system (ORDBMS). This system simply puts an object oriented front end on a relational database (RDBMS). When applications interface to this type of database, it will normally interface as though the data is stored as objects. However the system will convert the object information into data tables with rows and columns and handle the data the same as a relational database. Likewise, when the data is retrieved, it must be reassembled from simple data into complex objects.

**About Oracle Objects and Object Types:**

Oracle object types are user-defined data types that make it possible to model complex real-world entities such as customers and purchase orders as unitary entities-- "objects"--in the database.

Oracle object technology is a layer of abstraction built on Oracle's relational technology. New object types can be created from any built-in database types and any previously created object types, object references, and collection types. Metadata for user-defined types is stored in a schema that is available to SQL, PL/SQL, Java, and other published interfaces.

Object types and related object-oriented features such as variable-length arrays and nested tables provide higher- level ways to organize and access data in the database. Underneath the object layer, data is still stored in columns and tables, but you are able to work with the data in terms of the real-world entities--customers and purchase orders, for example--that make the data meaningful. Instead of thinking in terms of columns and tables when you query the database, you can simply select a customer.

Internally, statements about objects are still basically statements about relational tables and columns, and you can continue to work with relational data types and store data in relational tables as before. But now you have the option to take advantage of object-oriented features too. You can begin to use object-oriented features while continuing to work with most of your data relationally, or you can go over to an object-oriented approach entirely. For instance, you can define some object data types and store the objects in columns in relational tables. You can also create object views of existing relational data to represent and access this data according to an object model. Or you can store object data in object tables, where each row is an object.

**Advantages of Objects:**

In general, the object-type model is similar to the class mechanism found in C++ and Java. Like classes, objects make it easier to model complex, real-world business entities and logic, and the reusability of objects makes it possible to develop database applications faster and more efficiently. By natively supporting object types in the database, Oracle enables application developers to directly access the data structures used by their applications. No mapping layer is required between client-side objects and the relational database columns and tables that contain the data. Object abstraction and the encapsulation of object behaviors also make applications easier to understand and maintain.

Below are listed several other specific advantages that objects offer over a purely relational approach.

* Objects Can Encapsulate Operations Along with Data
* Objects Are Efficient
* Objects Can Represent Part-Whole Relationships

**Basic Components of Oracle Objects:**

Object-Relational Elements

Object-relational functionality introduces a number of new concepts and resources. These are briefly described in the following sections.

**Object Types:**

An object type is a kind of data type. You can use it in the same ways that you use more familiar data types such as NUMBER or VARCHAR2. For example, you can specify an object type as the data type of a column in a relational table, and you can declare variables of an object type. You use a variable of an object type to contain a value of that object type. A value of an object type is an instance of that type. An object instance is also called an object.

Object types also have some important differences from the more familiar data types that are native to a relational database:

A set of object types does not come ready-made with the database. Instead, you define the object types you want.

Object types are not unitary: they have parts, called attributes and methods.

You can think of an object type as a structural blueprint or template and an object as an actual thing built according to the template.

**Type Inheritance:**

You can specialize an object type by creating subtypes that have some added, differentiating feature, such as an additional attribute or method. You create subtypes by deriving them from a parent object type, which is called a super type of the derived subtypes.

Subtypes and super types are related by inheritance: as specialized versions of their parent, subtypes have all the parent's attributes and methods plus any specializations that are defined in the subtype itself. Subtypes and super types connected by inheritance make up a type hierarchy.

**Objects:**

When you create a variable of an object type, you create an instance of the type: the result is an object. An object has the attributes and methods defined for its type.

Because an object instance is a concrete thing, you can assign values to its attributes and call its methods.

**Design Analysis / Implementation Logic: Implementation:**

Object Tables

An object table is a special kind of table in which each row represents an object.

For example, the following statements create a person object type and define an object table for person objects:

CREATE TYPE person AS OBJECT ( name VARCHAR2(30), phone VARCHAR2(20) );

CREATE TABLE person\_table OF person;

You can view this table in two ways:

As a single-column table in which each row is a person object, allowing you to perform object-oriented operations

As a multi-column table in which each attribute of the object type person, namely name and phone, occupies a column, allowing you to perform relational operations

For example, you can execute the following instructions:

INSERT INTO person\_table VALUES (

"John Smith","1-800-555-1212" );

SELECT VALUE(p) FROM person\_table p WHERE p.name = "John Smith";

The first statement inserts a person object into person\_table, treating person\_table as a multi-column table. The second selects from person\_table as a single-column table, using the

VALUE function to return rows as object instances.

**Varrays:**

An array is an ordered set of data elements. All elements of a given array are of the same data type. Each element has an index, which is a number corresponding to the element's position in the array.

The number of elements in an array is the size of the array. Oracle allows arrays to be of variable size, which is why they are called varrays. You must specify a maximum size when you declare the array type.

For example, the following statement declares an array type:

CREATE TYPE prices AS VARRAY(10) OF NUMBER(12,2);

The VARRAYs of type PRICES have no more than ten elements, each of datatype NUMBER(12,2).

Creating an array type does not allocate space. It defines a datatype, which you can use as:

The datatype of a column of a relational table. An object type attribute. The type of a PL/SQL variable, parameter, or function return value.

A varray is normally stored in line, that is, in the same tablespace as the other data in its row. If it is sufficiently large, Oracle stores it as a BLOB.

A varray cannot contain LOBs. This means that a varray also cannot contain elements of a user-defined type that has a LOB attribute.

**Nested Tables:**

A nested table is an unordered set of data elements, all of the same datatype. It has a single column, and the type of that column is a built-in type or an object type. If the column in a nested table is an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type.

For example, in the purchase order example, the following statement declares the table type used for the nested tables of line items:

CREATE TYPE lineitem\_table AS TABLE OF lineitem;

A table type definition does not allocate space. It defines a type, which you can use as The datatype of a column of a relational table. An object type attribute.

A PL/SQL variable, parameter, or function return type.

When a column in a relational table is of nested table type, Oracle stores the nested table data for all rows of the relational table in the same storage table. Similarly, with an object table of a type that has a nested table attribute, Oracle stores nested table data for all object instances in a single storage table associated with the object table.

For example, the following statement defines an object

table for the object type PURCHASE\_ORDER:

CREATE TABLE purchase\_order\_table OF purchase\_order

NESTED TABLE lineitems STORE AS lineitems\_table;

The second line specifies LINEITEMS\_TABLE as the storage table for the LINEITEMS attributes of all of the PURCHASE\_ORDER objects in PURCHASE\_ORDER\_TABLE.

A convenient way to access the elements of a nested table individually is to use a nested cursor.

Testing The object person is created with name & phone no

Create the person\_table using the object person. Nested table purchase\_order table with puchase\_order and lineitems is created.

**Conclusion:**

Multivalued attributes and inheritance in ORDBMS is implemented.

**Problem Statement:**

T1,T2,T3,T4:

Find the Schema which contains the multivalued attribute, complex types and inheritance.

\

**EXPERIMENT NO. 4**

**TITLE: -** Implementation of Parallel Join and Sorting.

**AIM** **:** To study & Implement Parallel Join and Sorting

**Theory:**

**Parallel Sort:**

Suppose that we wish to sort a relation that resides on n disks D0, D1,...,Dn−1. If the relation has been range-partitioned on the attributes on which it is to be sorted, then, asnotedinSection18.2.2,we can sort each partition separately, and can concatenate the results to get the full sorted relation. Since the tuples are partitioned on n disks, the time required for reading the entire relation is reduced by the parallel access. If the relation has been partitioned in any other way, we can sort it in one of two ways:

1. We can range-partition it on the sort attributes, and then sort each partition separately. 2. We can use a parallel version of the external sort–merge algorithm.

**Range-Partitioning Sort:**

Range-partitioning sort works in two steps: ﬁrst range partitioning the relation, then sorting each partition separately. When we sort by range partitioning the relation, it is not necessary to range-partition the relation on the same set of processors or disks as those on which that relation is stored. Suppose that we choose processors P0, P1,...,Pm, where m < n, to sort the relation. There are two steps involved in this operation:

1. Redistribute the tuples in the relation, using a range-partition strategy, so that all tuples that lie within the ith range are sent to processor Pi, which stores the relation temporarily on disk Di. To implement range partitioning, in parallel every processor reads the tuples from its disk and sends the tuples to their destination processors. Each processor P0, P1,...,Pm also receives tuples belonging to its partition, and stores them locally. This step requires disk I/O and communication overhead.

2. Each of the processors sorts its partition of the relation locally, without interaction with the other processors. Each processor executes the same operation—namely, sorting—on a different dataset. (Execution of the same operation in parallel on different sets of data is called data parallelism.) The ﬁnal merge operation is trivial, because the range partitioning in the ﬁrst phase ensures that, for 1 ≤ i < j ≤ m, the key values in processor Pi are all less than the key values in Pj.

**Parallel Join:**

The join operation requires that the system test pairs of tuples to see whether they satisfy the join condition; if they do, the system adds the pair to the join output.Paralleljoinalgorithmsattempttosplitthepairstobetestedoverseveral processors. Each processor then computes part of the join locally. Then, the system collects the results from each processor to produce the ﬁnal result.

**Partitioned Join:**

For certain kinds of joins, such as equi-joins and natural joins, it is possible to partition the two input relations across the processors and to compute the join locally at each processor. Suppose that we are using n processors and that the relations to be joined are r and s.

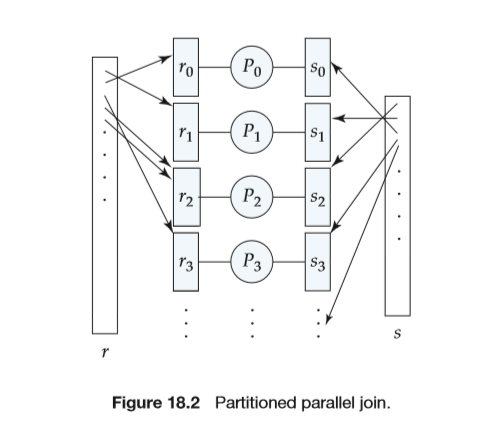
Partitioned join then works this way:

The system partitions the relations r and s each into n partitions, denotedr0,r1,...,rn−1 and s0,s1,...,sn−1. The system sends partitions ri and si to processor Pi, where their join is computed locally. The partitioned join technique works correctly only if the join is an equi-join (for example, r r.A=s.B s) and if we partition r and s by the same partitioning function on their join attributes. The idea of partitioning is exactly the same as that behind the partitioning step of hash join. In a partitioned join, however, there are two different ways of partitioning r and s:

• Range partitioning on the join attributes.

• Hash partitioning on the join attributes.

In either case, the same partitioning function must be used for both relations. For range partitioning, the same partition vector must be used for both relations. For hash partitioning, the same hash function must be used on both relations. Figure 18.2 depicts the partitioning in a partitioned parallel join. Once the relations are partitioned, we can use any join technique locally at each processor Pi to compute the join of ri and si. For example, hash join, merge join, or nested-loop join could be used. Thus, we can use partitioning to parallelize any join technique.



If one or both of the relations r and s are already partitioned on the join attributes (by either hash partitioning or range partitioning), the work needed for partitioning is reduced greatly. If the relations are not partitioned, or are partitioned on attributes other than the join attributes, then the tuples need to be repartitioned. Each processor Pi reads in the tuples on disk Di, computes for each tuple t the partition j to which t belongs, and sends tuple t to processor Pj. Processor Pj stores the tuples on disk Dj.

**Conclusion:** We have implemented the Parallel Sort and Join.

**Problem Statement:**

Write a Java program that consists implementation of Parallel Sort and Join.

**Experiment No. 5**

**Title:-**Implement two phase commit in distributed DBMS

**Theory:**

During normal execution, each site maintains a log, and the actions of a sub transaction are logged at the site where it executes. A commit protocol is followed to ensure that all sub transactions of a given transaction either commit or abort uniformly. The transaction manager at the site where the transaction originated is called the coordinatorfor the transaction; transaction managers at sites where its sub transactions execute are called subordinates**.** When the user decides to commit a transaction, the commit command is sent to the coordinator for the transaction. This initiates the 2PC protocol:

1. The coordinator sends a *prepare* message to each subordinate.

2. When a subordinate receives a *prepare* message, it decides whether to abort or commit its sub transaction. It force-writes an abort or preparelog record, and *then* sends a *no* or *yes* message to the coordinator.

3. If the coordinator receives *yes* messages from all subordinates, it force-writes a commit log record and then sends a *commit* message to all subordinates. If it receives even one *no* message, or does not receive any response from some subordinate for a specified time-out interval, it force-writes an abort log record, and then sends an *abort* message to all subordinates.

4. When a subordinate receives an *abort* message, it force-writes an abort log record, sends an *ack* message to the coordinator, and aborts the sub transaction. When a subordinate receives a *commit* message, it force-writes a commit log record, sends an *ack* message to the coordinator, and commits the sub transaction.

5. After the coordinator has received *ack* messages from all subordinates, it writes an end log record for the transaction.

The name *Two-Phase Commit* reflects the fact that two rounds of messages are exchanged: First a voting phase, then a termination phase, both initiated by the coordinator. The basic principle is that any of the transaction managers involved (including the coordinator) can unilaterally abort a transaction, whereas there must be unanimity to commit a transaction.

**Algorithm:-**

**At coordinator**

1. start
2. Create ServerSocket object and accept all subordinates request and connect with them.
3. Send a *prepare* message to each subordinate.
4. Receives *yes or no* messages from all subordinates.
5. If *yes* messages from all subordinates are received, write a commit log record and then send a *commit* message to all subordinates.
6. If *no* messages from all subordinates or even one are received, write an abort log record and then send an *abort* message to all subordinates.
7. Received *ack* messages from all subordinates, write an end log record for the transaction.
8. Stop.

**At subordinate**

1. start
2. Create Socket object and connect to coordinate.
3. Receives *prepare* messages from coordinate.
4. Send a *yes or no* message to coordinate.
5. Receives commit or abortmessage from coordinate.
6. Send a ack to coordinate.
7. Stop.

**Program**

**1.Coordinate**

import java.io.\*;

import java.net.\*;

class CoOrdinate

{

public static void main(String [] args)throws Exception

{

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

ServerSocket ss=new ServerSocket(4545);

System.out.println("\nEnter No. of Subordinate:");

int cnt=Integer.parseInt(br.readLine());

Socket []s=new Socket[cnt];

DataInputStream []dis=new DataInputStream[cnt];

DataOutputStream []dos=new DataOutputStream[cnt];

for(int i=0;i<cnt;i++)

{

s[i]=ss.accept();

dis[i]=new DataInputStream(s[i].getInputStream());

dos[i]=new DataOutputStream(s[i].getOutputStream());

}

String pr=new String("prepare");

System.out.println("Sending prepare message");

for(int i=0;i<cnt;i++){

dos[i].writeUTF(pr);

}

String []in=new String[cnt];

boolean flg=true;

for(int i=0;i<cnt;i++)

{

in[i]=dis[i].readUTF();

if(in[i].equals("no"))

flg=false;

}

if(flg==false)

{

pr="abort";

System.out.println("Sending abort msg");

for(int i=0;i<cnt;i++)

{

dos[i].writeUTF(pr);

}

}

else

{

pr="commit";

System.out.println("Sending commit msg");

for(int i=0;i<cnt;i++)

{

dos[i].writeUTF(pr);

}

for(int i=0;i<cnt;i++)

{

System.out.println("Receiving ack");

}

}

}

}

**2.Subordinate**

import java.io.\*;

import java.net.\*;

class SubOrdinate

{

public static void main(String [] args)throws Exception

{

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

Socket s=new Socket("localhost",4545);

DataInputStream dis =new DataInputStream(s.getInputStream());

DataOutputStream dos=new DataOutputStream(s.getOutputStream());

String out=new String();

String in=dis.readUTF();

System.out.println("Receiving "+in+" msg.");

System.out.println("SubOrdinate ready:'yes' or 'no':");

out=br.readLine();

System.out.println("Sending "+out);

dos.writeUTF(out);

in=dis.readUTF();

System.out.println("Reciving "+in+" msg");

if(in.equals("commit"))

{

out="ack";

System.out.println("Sending Ack");

}

}

}

**Output:-**

**At coordinator**

Enter No. of Subordinate:

4

Sending prepare message

Sending abort message

**At subrdinate1**

Receiving prepare msg.

Subordinate ready:’ yes' or 'no': yes

Sending yes

Receiving abort message.

**At subrdinate2**

Receiving prepare msg.

Subordinate ready:’ yes' or 'no': yes

Sending yes

Receiving abort message

**At subrdinate3**

Receiving prepare msg.

Subordinate ready:’ yes' or 'no': yes

Sending yes

Receiving abort message

**At subrdinate4**

Receiving prepare msg.

Subordinate ready:’ yes' or 'no': no

Sending no.

Receiving abort message

**EXPERIMENT NO. 6**

**TITLE: -** Implement vertical and horizontal fragmentation in distributed DBMS

**AIM** **:** To study & Implement fragmentation techniques in distributed systems.

**Theory:**

## What is fragmentation:

* The process of dividing the database into a smaller multiple parts is called as **fragmentation.**
* These fragments may be stored at different locations.
* The data fragmentation process should be carrried out in such a way that the reconstruction of original database from the fragments is possible.

## Types of data Fragmentation:

## 1. Horizontal data fragmentation:

Horizontal fragmentation divides a relation(table) horizontally into the group of rows to create subsets of tables.  
  
**Example:**  
Account (Acc\_No, Balance, Branch\_Name, Type).  
In this example if values are inserted in table Branch\_Name as Pune, Baroda, Delhi.  
  
**The query can be written as:**  
SELECT\*FROM ACCOUNT WHERE Branch\_Name= “Baroda”  
  
**Types of horizontal data fragmentation are as follows:**  
  
**1) Primary horizontal fragmentation**  
Primary horizontal fragmentation is the process of fragmenting a single table, row wise using a set of conditions.  
  
**Example:**

|  |  |  |
| --- | --- | --- |
| **Acc\_No** | **Balance** | **Branch\_Name** |
| A\_101 | 5000 | Pune |
| A\_102 | 10,000 | Baroda |
| A\_103 | 25,000 | Delhi |

For the above table we can define any simple condition like, Branch\_Name= 'Pune', Branch\_Name= 'Delhi', Balance < 50,000  
  
**Fragmentation1:**  
SELECT \* FROM Account WHERE Branch\_Name= 'Pune' AND Balance < 50,000  
  
**Fragmentation2:**  
SELECT \* FROM Account WHERE Branch\_Name= 'Delhi' AND Balance < 50,000  
  
**2) Derived horizontal fragmentation**  
Fragmentation derived from the primary relation is called as derived horizontal fragmentation.    
  
**Example:** Refer the example of primary fragmentation given above.  
  
**The following fragmentation are derived from primary fragmentation.**  
  
**Fragmentation1:**  
SELECT \* FROM Account WHERE Branch\_Name= 'Baroda' AND Balance < 50,000  
  
**Fragmentation2:**  
SELECT \* FROM Account WHERE Branch\_Name= 'Delhi' AND Balance < 50,000  
  
**3) Complete horizontal fragmentation**

* The complete horizontal fragmentation  generates a set of horizontal fragmentation, which includes every table of original relation.
* Completeness is required for reconstruction of relation so that every table belongs to at least one of the partitions.

**4) Disjoint horizontal fragmentation**  
The disjoint horizontal fragmentation generates a set of horizontal fragmentation in which no two fragments have common tables. That means every table of relation belongs to only one fragment.  
  
**5) Reconstruction of horizontal fragmentation**  
Reconstruction of horizontal fragmentation can be performed using  UNION operation on fragments.

## 2. Vertical Fragmentation:

Vertical fragmentation divides a relation(table) vertically into groups of columns to create subsets of tables.  
  
**Example:**

|  |  |  |
| --- | --- | --- |
| **Acc\_No** | **Balance** | **Branch\_Name** |
| A\_101 | 5000 | Pune |
| A\_102 | 10,000 | Baroda |
| A\_103 | 25,000 | Delhi |

**Fragmentation1:**  
SELECT \* FROM Acc\_NO  
  
**Fragmentation2:**  
SELECT \* FROM Balance  
  
**Complete vertical fragmentation**

* The complete vertical fragmentation generates a set of vertical fragments, which can include all the attributes of original relation.
* Reconstruction of vertical fragmentation is performed by using **Full Outer Join** operation on fragments.

**Conclusion:** We have implemented the fragmentation techniques in distributed systems.

**Problem Statement:**

Write a Java program that will implementation fragmentation techniques in distributed system.

**EXPERIMENT NO. 7**

**TITLE: -** Queries on PL/SQL Commands

**AIM** **:** To study & make use of PL/SQL commands to perform different operations in dbms.

**Theory:**

## Creating a Function:

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows −

CREATE [OR REPLACE] FUNCTION function\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END [function\_name];

Where,

* *function-name* specifies the name of the function.
* [OR REPLACE] option allows the modification of an existing function.
* The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
* The function must contain a **return** statement.
* The *RETURN* clause specifies the data type you are going to return from the function.
* *function-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone function.

### Example:

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the [PL/SQL Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter −

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

CREATE OR REPLACE FUNCTION totalCustomers

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(\*) into total

FROM customers;

RETURN total;

END;

/

When the above code is executed using the SQL prompt, it will produce the following result −

Function created.

## Calling a Function:

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block −

DECLARE

c number(2);

BEGIN

c := totalCustomers();

dbms\_output.put\_line('Total no. of Customers: ' || c);

END;

/

When the above code is executed at the SQL prompt, it produces the following result −

Total no. of Customers: 6

PL/SQL procedure successfully completed.

### Example:

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

DECLARE

a number;

b number;

c number;

FUNCTION findMax(x IN number, y IN number)

RETURN number

IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

dbms\_output.put\_line(' Maximum of (23,45): ' || c);

END;

/

When the above code is executed at the SQL prompt, it produces the following result −

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

## Creating a Procedure:

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

CREATE [OR REPLACE] PROCEDURE procedure\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

< procedure\_body >

END procedure\_name;

Where,

* *procedure-name* specifies the name of the procedure.
* [OR REPLACE] option allows the modification of an existing procedure.
* The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
* *procedure-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone procedure.

### Example:

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

When the above code is executed using the SQL prompt, it will produce the following result −

Procedure created.

## Executing a Standalone Procedure

A standalone procedure can be called in two ways −

* Using the **EXECUTE** keyword
* Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

EXECUTE greetings;

The above call will display −

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block −

BEGIN

greetings;

END;

/

The above call will display −

Hello World

PL/SQL procedure successfully completed.

## Deleting a Standalone Procedure:

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

DROP PROCEDURE procedure-name;

You can drop the greetings procedure by using the following statement −

DROP PROCEDURE greetings;

## Parameter Modes in PL/SQL Subprograms:

The following table lists out the parameter modes in PL/SQL subprograms −

|  |  |
| --- | --- |
| **S.No** | **Parameter Mode & Description** |
| 1 | **IN**  An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**. |
| 2 | **OUT**  An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**. |
| 3 | **IN OUT**  An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.  The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.** |

### IN & OUT Mode Example:

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

DECLARE

a number;

b number;

c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

IF x < y THEN

z:= x;

ELSE

z:= y;

END IF;

END;

BEGIN

a:= 23;

b:= 45;

findMin(a, b, c);

dbms\_output.put\_line(' Minimum of (23, 45) : ' || c);

END;

/

When the above code is executed at the SQL prompt, it produces the following result −

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

**Conclusion:** We have implemented the PL/SQL commands.

**Problem Statement:**

Write Function and Procedure in PL/SQL.