**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**10211CS207- DATABASE MANAGEMENT SYSTEMS**

**LAB MANUAL**

**Academic Year: 2025-2026**

**SUMMER SEMESTER**

**SLOT – S1 + L4**

**Tamil Nadu Cricket Association**

**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**10211CS207- DATABASE MANAGEMENT SYSTEMS**

**LAB MANUAL**

**Academic Year: 2025-2026**

**SUMMER SEMESTER**

**SLOT – S1 + L4**

**CASE of TASK: Tamil Nadu Cricket Association**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Prepared by | Verified by | Authorized by |
| Signature |  |  |  |
| Name with TTS ID | Dr. N. Rajkumar  TTS 1577 |  |

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.NO.** | **PARTICULARS** | **PAGE NO** |
| 1 | Syllabus | 4 |
| 2 | Course Plan | 10 |
| 3 | Task Plan | 13 |
| 4 | SQL: Basics and Implemantation of DDL commands | 20 |
| 5 | Implementation of DML commands | 24 |
| 6 | Nested Queries and Join Queries | 27 |
| 7 | Constraints | 35 |
| 8 | Virtual Table (View) | 38 |
| 9 | PL/SQL | 43 |
| 10 | Triggers | 50 |
| 11 | Task Master Manual | 53 |
| 12 | Viva- voce Questions | I |



**SYLLABUS**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **COURSE CODE** | **COURSE TITLE** | **L** | **T** | **P** | **C** |
| **10211CS207** | **DATABASE MANAGEMENT SYSTEMS** | **3** | **0** | **2** | **4** |

**A.Preamble**

Databases form the backbone of all major applications today – tightly or loosely coupled, intranet or internet based, financial, social, administrative, and so on. Structured Database Management Systems (DBMS) based on relational and other models have long formed the basis for such databases. While DBMS differ in the details, they share a common set of models, design paradigms via a Structured Query Language (SQL) or NoSQL. More specifically, this course introduces relational data models; entity-relationship modeling, SQL, data normalization, database design and modern databases asd well lightweight databases.

**B.Prerequisite Course**

10210CS101 – Problem Solving using C

**C.Course Objectives**

Learners are exposed to:

* Design a database at basic level like Entity-Relationship modeling, Relational Modeling.
* Devise a database using Armstrong axioms, functional dependencies and views.
* Acquire knowledge of Query processing and concurrency transaction control.
* Learn an implementation issues with nontraditional databases.



**D.Course Outcomes**

Upon the successful completion of the course, students will be able to:

|  |  |  |
| --- | --- | --- |
| **CO**  **No’s** | **Course Outcomes** | **K – Level/ S - Level** |
| **CO1** | Apply basic database designing methodologies | **K3/S3** |
| **CO2** | Design a database schema for a given problem | **K3/S3** |
| **CO3** | Apply normalization for the given database. | **K3/S3** |
| **CO4** | Use the properties of transaction and recovery management. | **K2/S3** |
| **CO5** | Implement CRUD operations in modern database. | **K3/S3** |
| **Knowledge Level (Based on revised Bloom’s Taxonomy)**  K1-Remember K2-Understand K3-Apply K4-Analyze K5-Evaluate K6-Create  S1-Imitate, S2-Perform, S3-Precision, S4-Articulation, and S5-Naturalization | | |



**E.Correlation of Cos with Program Outcomes and Programme Specific Outcomes**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Cos** | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PSO1** | **PSO2** | **PSO3** |
| **CO1** | **2** | **2** | **2** |  |  |  |  |  |  |  |  | **2** |  | **2** |
| **CO2** |  | **3** | **3** | **2** | **3** |  |  |  |  |  |  | **3** | **3** | **3** |
| **CO3** |  | **3** | **3** |  | **2** |  |  |  |  |  |  | **2** | **2** | **2** |
| **CO4** |  | **2** | **2** |  |  |  |  |  |  |  |  |  |  | **2** |
| **CO5** |  | **2** |  | **2** | **3** |  |  |  |  |  |  |  | **2** | **2** |

High- 3; Medium-2; Low-1



**F.Course Contents**

**Unit 1 Introduction to DBMS L–9 Hours**

Purpose of Database System – Database Schema and Instances- Views of data – Database Languages - Database System Architecture – Database users and Administrator –Distributed Databases-DDB System Architecture-Database models-Entity–Relationship model – E-R Diagrams - Introduction to relational databases –Structure of relational databases- Relational model-Basics- From E/R diagrams to Database Design- Hierarchical model-Tree-structure diagrams- data retrieval-update facility-virtual records-mapping of hierarchies to files-Network Model- Data-structure diagrams-DBTG model- data retrieval-Find-Get-Set processing-update facility-Mapping network to files. Case study: Conference Management system.

**Unit 2 Database Design and Query Optimization L–9 Hours**

Keys and Integrity Constraints - Relational Algebra – Relational Calculus-Tuple –Structured Query language( SQL) Basic and additional Operations –Database Languages Revisited –create-alter- update-insert-rename-truncate-drop-select query commands-grant-revoke-rollback-commit control commands-Operators-aggregate functions-string-indent-date functions-set-union-union all-aliasing-Nested Queries & Join Queries–Join-Natural-inner-equi-cross/cartesian-outer-left-right-full outer join-Common Table Expressions-limit function in statistics-summarizing-Embedded SQL- Indexes and Triggers –compound- Views Definitions -Materialized view-Assertions-Query Processing and Optimization-Heuristics and cost estimates in Query Optimization-Tuning. Case study: Restaurant chain for network model.

**Unit 3 Functional Dependencies and Normalization L–9 Hours**

Introduction and problem of data redundancy-Features of good Relational database design- Functional Dependencies - Normalization – First Normal Form-minimal cover-canonical cover-Decomposition- Second Normal Form- Third Normal Form –controversies with SQL in normalization-Advanced Normalization -Boyce/Codd Normal Form, Fourth Normal Form and Fifth Normal Form- Dependencies preservation-Denormalization-Case Studies of database system like Air-cargo services**.**

**Unit 4 Transaction and Concurrency L–9 Hours**

Transaction Concepts –Schedules- Transaction States - Concurrent Execution- Serializability- -Two Phase locking Protocol –Timestamp ordering Protocol-Validation Based Protocol-Multiple Granularity-Intention lock modes-Multiple granularity locking scheme-Replication in Distributed databases-Deadlock-Handling-Prevention-Wait-diescheme- wound-wait scheme- Detection-Wait-for graph-Phantom phenomenon-Distributed deadlock detection-Index locking-weak levels of concurrency- Concurrency in index structures- B+ Tree crabbing- -Types of Failure-Recoverability -Recovery and Prevention- Log based recovery-undo-redo-checkpoint-Recovery Algorithm-Transaction rollback with Logical undo- Recovery Algorithm with Logical Undo-ARIES Recovery Algorithm. Case study: Instances and Log recovery process in Oracle and Microsoft Access.

**Unit 5 Storage and Modern Databases L–9 Hours**

RAID storage -Modern databases-XML Databases- CAP Theorem-Datalog-Basic structure-Syntax-semantics-Relational operation in Datalog-Recursion in Datalog-Power of Recursion-Monotonicity-Non-monotonicity-NoSQL-Data models-Key value store, column families-Document Databases-MongoDB-Introduction to MongoDB- CRUD operations-Create-Insert-Update-Query-Removing documents-Sharding- Introduction to Mongoose-core concepts-extending models-CRUD operations with mongoose. Case study: Designing local library in Amazon using mongoose-SQLite-Building and Installing SQLite-DDL-DML-TCL-Select-compound select- alternate Join- JOIN..ON, JOIN..USING-Database design-SQL functions and SQLite extensions-Virtual Tables and Modules.

**Total: 45 Hours (L+T)**

**G.Laboratory Experiments Total: 30 Hours**

**Part- 1**

**Task 1: Conceptual Design after FTR**

Using basic database design methodology and ER modeler, design Entity Relationship Diagram by satisfying the following sub tasks:

**1. a** Identifying the entities.

**1. b** Identifying the attributes.

**1. c** Identification of relationships, cardinality, type of relationship.

**1. d** Reframing the relations with keys and constraints.

**Task 2: Generating design of other traditional database model**

Creating Hierarchical /Network model of the database by enhancing the sound abstract data by performing following tasks using forms of inheritance:

**2. a** Identify the specificity of each relationship, find and form surplus relations.

**2.b**Check is-a hierarchy/ has-a hierarchy and performs generalization and/or specialization relationship.

**2. c** Find the domain of the attribute and perform a check constraint to the applicable.

**2. d** Rename the relations.

**2. e** Perform SQL Relations using DDL, DCL commands.

**Task 3: Using Clauses, Operators and Functions in queries**

Perform the query processing on databases for different retrieval results of queries using DML, DRL operations using aggregate, date, string, indent functions, set clauses and operators.

**Task 4: Using Functions in queries and writing subqueries:**

Perform the advanced query processing and test its heuristics using the designing of optimal correlated and nested subqueries such as finding summary statistics.

**Task 5: Writing Join Queries, equivalent, and/or recursive queries**

Perform the advanced query processing and test its heuristics using the designing of optimizing complex queries and their equivalence queries

**TASK 6: Procedures, Function and Loops**:

Programming using PL/SQL Procedures, Functions and loops on Number theory and business scenarios like.

**TASK 7**: **Triggers, Views and Exceptions**

Conduct events, views, exceptions on CRUD operations for restricting phenomenon.

**TASK 8**: **CRUD operations in Document databases**

Perform Mongoose using NPM design on MongoDB designing document database and performing CRUD operations like creating, inserting, querying, finding, removing operations.

**TASK 9: CRUD operations in Graph databases**

Perform GraphQL/Neo4jgraph space design for recommendation engines. Also perform CRUD operations like creating, inserting, querying, finding, deleting operations on graph spaces.

**Task 10**: **Normalizing databases using functional dependencies up to BCNF**

Upon relational tables created in task-2, perform normalization up to BCNF based on given Dependencies as following for the assumed relations specified below.

**10. a** Apply the functional dependency, normalize to 1NF

**10. b** Normalize the relations using FD+ and α+

**10. c** Find the minimal cover, canonical cover.

**10. d** Normalize to 2NF, add/alter constraints if necessary.

**10. e** Normalize to BCNF, add/alter constraints if necessary.

**10. f** Normalize to 3NF, add/alter constraints if necessary.

**Task 11: Menus, Forms and Reports:**

For an application, creating and debugging Menus, Forms and reports using Oracle Forms and Report Builder or SQL server management studio or SQLite or NetBeans

.

**Task 12: Micro Project:**

Develop micro projects based on business scenarios and use cases.

**Part-II**

**Use cases:**

**Use Case-1: Building a Cart analysis for Myph**

Myph has just launched their brand new phone range to the eager reception of the consumer market cart analysis. The product's data model has a unique menu that identifies the product, title, description, a stock quantity, and pricing information about the item. All products have categories. To be able to provide a list of all the products in a category, amend the data model with a collection of documents for each category and contain the path for that category in the category tree. Use cart analysis in developing different consumer selection options. Would this answer outlier selection in cart i.e., surplus selections? Is Relational database application can answer these transactions? How recovery is made through carting and commerce?

**Use Case-2: Indexing various devices in IoT platform**

A generic IoT platform required support for data from a wide range of devices, some of which could not be envisaged while developing the platform. The proficient work necessitated a data storage mechanism that could handle data from different types of devices. Indexing support makes it easy to pull data using a single index or multiple indexes such as device id with location id. Records for a particular device in different locations are easily accessed. Common parameters like temperature from different types of devices and their records are retrieved fast through these indexes. How the application could lead to the choice of JSON-based document database, MongoDB? Assume or create JSON script in support of this.

**Use Case-3: A gift coupon application handles offers payment-related Information.**

Choose the database system, a relational database, for its capability to scale horizontally while keeping the ACID property. The last was particularly important because transactional data was being handled. Eventual consistency as offered by other databases was not suitable. Going with DB and cluster gives the opportunity to scale horizontally for a large number ofwrites and reads without compromising the ACID and transactional capability required by the application. Will it transact and lead to no deadlock, keeping all relational tables normalized? If so till what normal form do they sustain to offer a gift coupon application?

**Use case- 4: Army Supply chain, Bill of Materials and Maintenance Cost Management**

The nation’s armed forces, the support for more than one million soldiers and about 200,000 civilian staff. Each of these staff members relies on multiple pieces of equipment, from helicopters and armoured vehicles to small arms and radios, to complete their missions. With maintenance, operation and support costs of equipment representing as much as 80% of total lifecycle costs, it’s imperative that the Defence ministry track and analyse equipment maintenance costs including changing historical data sources Dimension with more flexibility like graph databases and to be given richer analysis like forecast replacement parts with the location and climate, mean time to failure rates, logistics and requirement processes. Is answerable the vital “what-if” questions such as the cost of deploying certain forces and supporting equipment to a new war zone? Could the model perform multi-dimensional cost comparison and trend analysis? Will the solution promises how the data management in unpredictable maintenance costs?

**H.Learning Resources**

1. **Text Books:**
2. Abraham Silberschatz, Henry F. Korth and S. Sudharshan, “Database System Concepts”, Seventh Edition, Tata McGraw Hill, 2019.[ Unit1-5]
3. Shannon Bradshaw, E Brazil, Kristina Chodorow, "MongoDB: The Definitive Guide - Powerful and Scalable Data Storage", Third Edition, Shroff/O'Reilly Inc., January 2020.[Unit-5]
4. AgusKurniawin, “Python and SQLite Development”, First Edition, PE Press, January 2021. [Unit-5]
5. **Reference Books:**
6. Raghu Ramakrishnan et al, “Database Management Systems”, Third Edition, McGraw Hill, 2014.
7. Elmasri Ramez, Navathe S, “Fundamentals of Database System”, Seventh Edition, Pearson, 2017.
8. J.D.Ullmann et al, "Database Systems: The Complete Book", Second Edition, Pearson Ed, Inc, 2009.
9. **Online References:**
10. “Database Management System Part-1&11” Mar.25, 2019. Accessed on Feb.20, 2021[Online].Available:https://infytq.onwingspan.com/en/viewer/web\_module/lex\_auth\_0130542736573808641006\_shared?collectionId=lex\_auth\_01275806667282022456\_shared&collectionType=Course.
11. “Database Design”, Dec 31, 2009. Accessed on May05,2021[Online]. Available: <https://nptel.ac.in/courses/106/106/106106093/>.
12. “Designing local library models”, Accessed on: May 05, 2021[online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\_Nodejs/mongoose.

**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**LESSON PLAN**

**Programme :** B.Tech – Computer Science and Engineering

**Course Code / Course Name :** 10211CS207 / Database Management Systems

**Year / Semester :** 2025-26 / Summer

**Course Handling Faculty Name :** Dr. N. Rajkumar

**Slot No :** S1 + L4

**Course Coordinator Name :** Dr. N. K. Senthil Kumar&Mrs. S. Hanna

1. **Preamble:**

Databases form the backbone of all major applications today – tightly or loosely coupled, intranet or internet based, financial, social, administrative, and so on. Structured Database Management Systems (DBMS) based on relational and other models have long formed the basis for such databases. While DBMSs differ in the details, they share a common set of models, and design paradigms via a Structured Query Language (SQL) or NoSQL. More specifically, this course introduces relational data models; entity-relationship modeling, SQL, data normalization, database design, modern databases, and well lightweight databases.

1. **Prerequisite Courses:**

|  |  |  |
| --- | --- | --- |
| Sl. No | Course Code | Course Name |
| 1 | 10211CS102 | [Data](#s) Structures |

1. **Course Outcomes:**

Upon the successful completion of the course, students will be able to:

|  |  |  |
| --- | --- | --- |
| **CO**  **Nos.** | **Course Outcomes** | **Level of learning domain (Based on revised Bloom’s)** |
| CO1 | Apply the various database design methodologies | **K3** |
| CO2 | Design a database schema for a given problem | **K3** |
| CO3 | Apply normalization for the given databases. | **K3** |
| CO4 | Use the properties of transaction and recovery management. | **K2** |
| CO5 | Apply CRUD operations in modern databases. | **K3** |

1. **Correlation of COs with POs :**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **COs** | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** | **PSO 1** | **PSO2** | **PSO 3** |
| CO1 | **2** | **2** | **2** |  |  |  |  |  |  |  |  |  | **2** |  | **2** |
| CO2 |  | **3** | **3** | **2** | **3** |  |  |  |  |  |  |  | 3 | **3** | **3** |
| CO3 |  | **3** | **3** |  | **2** |  |  |  |  |  |  |  | 2 | **2** | **2** |
| CO4 |  | **2** | **2** |  |  |  |  |  |  |  |  |  |  |  | **2** |
| CO5 |  | **2** |  | **2** | **3** |  |  |  |  |  |  |  |  | **2** | **2** |

3 - High; 2 - Medium; 1 - Low

1. **Learning Resources:**

**i.Text Books:**

1. Abraham Silberschatz, Henry F. Korth and S. Sudharshan, “Database System Concepts”, Seventh Edition, Tata McGraw Hill, 2019.[ Units-1,2,3,4,5]
2. Shannon Bradshaw, E Brazil, Kristina Chodorow, "MongoDB: The Definitive Guide - Powerful and Scalable Data Storage", Third Edition, Shroff/O'Reilly Inc., January 2020.[Unit-5]
3. AgusKurniawin, “Python and SQLite Development”, First Edition, PE Press, January 2021. [unit-5]

**ii.Reference Books:**

1. Raghu Ramakrishnan et al, “Database Management Systems”, Third Edition, McGraw Hill, 2014.
2. Elmasri Ramez, Navathe S, “Fundamentals of Database System”, Seventh Edition, Pearson, 2017.
3. J.D.Ullmann et al, "Database Systems: The Complete Book", Second Edition, Pearson Ed, Inc, 2009.

**iii.Online References:**

1. “Database Management System Part-1&11” Mar.25, 2019. Accessed on Feb.20,   2021[Online].Available:https://infytq.onwingspan.com/en/viewer/web\_module/lex\_auth\_0130542736573808641006\_shared?collectionId=lex\_auth\_01275806667282022456\_shared&collectionType=Course.
2. “Database Design”, Dec 31, 2009. Accessed on May05,2021[Online]. Available: https://nptel.ac.in/courses/106/106/106106093/.
3. “Designing local library models”, Accessed on: May 05, 2021[online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\_Nodejs/mongoose.

**LESSON PLAN**

**LABORATORY**

|  |  |  |  |
| --- | --- | --- | --- |
| **Task No.** | **Task Description** | **Teaching Methods / Tools** | **Planned Date** |
|  | Conceptual Design after FTR | Discussion, ERWIN,  SQL/ SQLite | 2 labs/ 4 Hours |
|  | Generating design of other traditional database model | Discussion, ERWIN,  SQL/ SQLite | 1 lab/ 2 hours |
|  | Using Clauses, Operators and Functions in queries | Demo, SQL/ SQLite/ MySQL | 2 labs/ 4 hours |
|  | Using Functions in queries and writing sub queries | Demo, SQL/ SQLite/ MySQL | 2 labs/ 4 hours |
|  | Writing Join Queries, equivalent, and/or recursive queries | Demo, SQL/ SQLite/ MySQL | 1 lab/ 2 Hours |
|  | Procedures, Function and Loops | Demo, SQL/ SQLite/ MySQL | 1 lab/ 2 Hours |
|  | Triggers, Views and Exceptions | Demo, SQL/ SQLite/ MySQL | 1 lab/ 2 hours |
|  | CRUD operations in Document databases | Demo, MongoDB | 1 lab/ 2 hours |
|  | CRUD operations in Graph databases | Demo, Neo4j | 1 lab/ 2 hours |
|  | Normalizing databases using functional dependencies up to BCNF | Discussion, GU/Table Normalization Tool | 2 labs/ 4 hours |
|  | Menus, Forms and Reports | Demo, NetBeans/ SQL Developer | 1 lab/ 2 hours |
|  | **Part-II : Use cases:** Micro Project | NetBeans/ MySQL, SQL, SQLite |  |

**Course Handling Faculty Course Coordinator**

**Dr. N. Rajkumar / TTS1577 Dr. N. K. Senthil Kumar / Mrs. S. Hannah**

**Date:**

**Note:**

* **PPT –** Power Point Presentations
* **Demo -** Demonstration

## Introduction of Oracle database

Oracle database is a relational database management system. It is also called **OracleDB**, or simply **Oracle**. It is produced and marketed by **Oracle Corporation**. It was created in **1977** by **Lawrence Ellison** and other engineers. It is one of the most popular relational database engines in the IT market for storing, organizing, and retrieving data.

Oracle database was the first DB that designed for **enterprise grid computing** and data warehousing. Enterprise grid computing provides the most flexible and cost-effective way to manage information and applications. It uses SQL queries as a language for interacting with the database.

Editions of Oracle database

Oracle database is compatible with a wide range of platforms such as Windows, UNIX, Linux, and macOS. It supports several operating systems like IBM AIX, HP-UX, Linux, Microsoft Windows Server, Solaris, SunOS, macOS, etc. In the late **1990s**, Oracle began supporting open platforms like GNU/Linux.

**The following is a list of Oracle database editions in order of priority:**

* **Enterprise Edition:** It is the most robust and secure edition. It offers all features, including superior performance and security.
* **Standard Edition:** It provides the base functionality for users that do not require Enterprise Edition's robust package.
* **Express Edition (XE):** It is the lightweight, free and limited Windows, and Linux edition.
* **Oracle Lite:** It is designed for mobile devices.
* **Personal Edition:** It's comparable to the Enterprise Edition but without the Oracle Real Application Clusters feature.

The Oracle Corporation

Oracle Corporation is the largest software company to develop and markets computer software applications for business. The company is best known for its Oracle database products and, more recently, cloud products and services. Its relational database was the first to support [SQL](https://www.javatpoint.com/sql-tutorial), which has since become the industry standard.

Oracle database is one of the most trusted and widely used relational database engines. The biggest rival of Oracle database is Microsoft's SQL Server.

History of Oracle

**Oracle Corporation was founded by Lawrence Ellison (Larry Ellison), Bob Miner, Ed Oates, and Bruce Scott in August 1977**. They have a lot of experience in building database programs for several companies and builds their first project (a special database program) for the **CIA (Central Intelligence Agency).** Oracle was named after "Project Oracle," a project for one of their clients named Central Intelligence Agency, and the company that created Oracle was called **Systems Development Labs (SDL)**. Systems Development Labs was renamed Relational Software Inc. (RSI) in 1978 to expand their market for the new database. They had again changed the name of the company from RSI to Oracle Systems Corporation in 1982.

The first commercially available RDBMS named **Oracle V2 (Version 2)** was built using PDP-11 assembler language (SQL-based RDBMS). Although they already developed a commercial RDBMS in 1977, it wasn't available for purchase until 1979, when Oracle version 2 was released.

**In 1983**, Oracle database portable version named "Oracle version 3" was released. This version was written in the C programming language. It was the first relational database that can run in mainframes, minicomputers, PCs, or any hardware with a C compiler. It also supports SQL queries and transactions execution.

The other subsequent versions are:

* **In 1984**, Oracle 4 was released that supports Transactions [Commit/Rollback], export/import utilities, and the report writer.
* **In 1985**, Oracle 5 was released, which provides support for Client-Server Architecture. This new feature has the capability to connect the client's software to a database server through a network.
* **In 1989**, Oracle 6 added support for PL/SQL language. It also comes with new features such as OLTP high-speed systems, hot backup capability, and row-level locking.
* **In 1992**, Oracle 7 was released. This version comes in the market as a result of four years of hard work and two years of customer testing. It added some exciting features and capabilities in the area of security, administration, development, and performance.
* **In 1997**, Oracle 8 was released. This version comes with the support of ORDBMS that was designed to work with Oracle's network computer (NC). It also added support for Java, HTML, and OLTP.
* **In 1998**, Oracle 8i was released. Here 'I' stands for Internet. It was the first database version that added support for Web technologies such as Java and HTTP.
* **In 2001**, Oracle 9i was released with 400 new features such as XML, RAC (Real Application Clusters), etc. These features reduce database size and provide high availability & enhanced performance.
* **In 2003**, Oracle 10g was released with grid computing technology means grid. It was the first version that supports 64-bit LINUX OS.
* **In 2006**, Oracle 11g was released. This version comes with new features such as Oracle Database Replay, Transaction Management using Log Miner, Virtual Column Partitioning, Case sensitive passwords, Online Patching, Parallel Backups on the same file using RMAN, and many others.
* **In July 2014**, Oracle 12C was released with Cloud support.
* **In Feb 2018**, Oracle 18 C was released. This version was the world's first autonomous database.

## Oracle Database Features

Oracle database manages data with the help of an open, complete, and integrated approach. The following are features that complete the demand for powerful database management:

**Availability:** It is never offline or out of service that means supported 24\*7 availability of the database. It provides high availability of databases because of the Oracle Data Guard functionality. This functionality allows using of the secondary database as a copy of the primary database during any failure. As a result, all normal processes such as backups and partial failures do not interrupt the database from being used.

**Security:** Oracle has a mechanism for controlling and accessing the database to prevent unauthorized access. It provides high security because of the Oracle Advanced Security features. It offers two solutions to protect databases that are TDE (Transparent Data Encryption) and Data Redaction. TDE supports data encryption both at the source and after export. Redaction is performed at the application level. Oracle has some other security features like Oracle Database Vault that regulates user privileges and Oracle Label Security.

**Scalability:** It provides features like RAC (Real Application Cluster) and Portability, which makes an Oracle database scalable based on usage. In a clustered environment, it includes capabilities such as rolling instance migrations, performing upgrades, maintaining application continuity, quality of service management, etc.

**Performance:** Oracle provides performance optimization tools such as Oracle Advanced Compression, Oracle Database In-Memory, Oracle Real Application Testing, and Oracle Times Ten Application-Tier Database Cache. Their main objective is to improve system performance to the highest possible level.

**Analytics:** Oracle has the following solutions in the field of analytics:

* **OLAP (Oracle Analytic Processing):** It is an implementation of Oracle for doing complicated analytical calculations on business data.
* **Oracle Advanced Analytics:** It is a technical combination of Oracle R Enterprise and Oracle Data Mining that assists customers in determining predictive business models through data and text mining, as well as statistical data computation.

**Management:** Oracle Multitenant is a database management tool that combines a single container database with many pluggable databases in a consolidated design.

Benefits of Oracle Database

The following are the main advantages of an Oracle database:

**Performance:** Oracle has procedures and principles that help us to get high levels of database performance. We can increase query execution time and operations with the use of performance optimization techniques in its database. This technique helps to retrieve and alter data faster.

**Portability:** The Oracle database can be ported on all different platforms than any of its competitors. We can use this database on around 20 networking protocols as well as over 100 hardware platforms. This database makes it simple to write an Oracle application by making changes to the OS and hardware in a secure manner.

**Backup and Recovery:** It is always better to take a proper backup of your entire oracle online backup and recovery. The Oracle database makes it easy to accomplish recovery quickly by using the. RMAN (Recovery Manager) functionality. It can recover or restore database files during downtime or outages. It can be used for online backups, archived backups, and continuous archiving. We can also use SQL\* PLUS for recovery, which is known as user-managed recovery.

**PL/SQL:** One of the greatest benefits of using the Oracle database is to support PL/SQL extension for procedural programming.

**Multiple Database:** Oracle database allows several database instances management on a single server. It provides an instance caging approach for managing CPU allocations on a server hosting database instances. The database resource management and instance caging can work together to manage services across multiple instances.

**Flashback Technology:** This advantage comes with the recent Oracle version. It allows us to recover those data that are incorrectly deleted or lost by human errors like accidental deletion of valuable data, deleting the wrong data, or dropping the table.

Disadvantages of Oracle Database

The following are the disadvantages of the Oracle database:

**Complexity:** Oracle is not recommended to use when the users are not technically savvy and have limited technical skills required to deal with the Oracle Database. It is also not advised to use if the company is looking for a database with limited functionality and easy to use.

**Cost:** The price of Oracle products is very high in comparison to other databases. Therefore users are more likely to choose other less expensive options such as MS SQL Server, MySQL, etc.

**Difficult to manage:** Oracle databases are often much more complex and difficult in terms of the management of certain activities.

**DATA DEFINITION, CONSTRAINTS, AND SCHEMA CHANGES**

Used to CREATE, ALTER, and DROP the descriptions of the database tables (relations)

**Data Definition in SQL**

**CREATE, ALTER and DROP**

table…………………………………….……relation row……………………………………..…….tuple column………………………………….……attribute

**DATA TYPES**

Numeric: NUMBER, NUMBER(s,p), INTEGER, INT, FLOAT, DECIMAL

Character: CHAR(n), VARCHAR(n), VARCHAR2(n), CHAR VARYING(n)

Bit String: BLOB, CLOB

Boolean: true, false, and null

Date and Time: DATE (YYYY-MM-DD) TIME( HH:MM:SS)

Timestamp: DATE + TIME

USER Defined types

**CREATE SCHEMA**

Specifies a new database schema by giving it a name

Ex: CREATE SCHEMA COMPANY AUTHORIZATION Jsmith;

**CREATE TABLE**

Specifies a new base relation by giving it a name, and specifying each of its attributes and their data types

Syntax of CREATE Command:

**CREATE TABLE <***table name>* ( <Attribute *A*1> <Data Type *D*1> [< Constarints>],

<Attribute *A*2> <Data Type *D*2> [< Constarints>],

*…….*

<Attribute *A*n> <Data Type *D*n> [< Constarints>],

[<integrity-constraint1>, <integrity-constraint k> ] );

- A constraint NOT NULL may be specified on an attribute

A constraint NOT NULL may be specified on an attribute

Ex: CREATE TABLE DEPARTMENT (

DNAME VARCHAR(10) NOT NULL,

DNUMBER INTEGER NOT NULL,

MGRSSN CHAR(9), MGRSTARTDATE CHAR(9) );

Specifying the unique, primary key attributes, secondary keys, and referential integrity constraints (foreign keys).

Ex: CREATE TABLE DEPT (

DNAME VARCHAR(10) NOT NULL,

DNUMBER INTEGER NOT NULL,

MGRSSN CHAR(9),

MGRSTARTDATE CHAR(9),

PRIMARY KEY (DNUMBER),

UNIQUE (DNAME),

FOREIGN KEY (MGRSSN) REFERENCES EMP(SSN));

We can specify RESTRICT, CASCADE, SET NULL or SET DEFAULT on referential integrity constraints (foreign keys)

Ex: CREATE TABLE DEPT (

DNAME VARCHAR(10) NOT NULL,

DNUMBER INTEGER NOT NULL,

MGRSSN CHAR(9), MGRSTARTDATE CHAR(9),

PRIMARY KEY (DNUMBER),

UNIQUE (DNAME),

FOREIGN KEY (MGRSSN) REFERENCES EMP

ON DELETE SET DEFAULT ON UPDATE CASCADE);

**DROP TABLE**

Used to remove a relation (base table) and its definition.

The relation can no longer be used in queries, updates, or any other commands since its description no longer exists

**Example:** DROP TABLE DEPENDENT;

**ALTER TABLE:**

Used to add an attribute to/from one of the base relations drop constraint -- The new attribute will have NULLs in all the tuples of the relation right after the command is executed; hence, the NOT NULL constraint is *not allowed* for such an attribute.

**Example:** ALTER TABLE EMPLOYEE ADD JOB VARCHAR2 (12);

The database users must still enter a value for the new attribute JOB for each EMPLOYEE tuple. This can be done using the UPDATE command.

**DROP A COLUMN (AN ATTRIBUTE)**

ALTER TABLE COMPANY.EMPLOYEE DROP ADDRESS CASCADE; All constraints and views that reference the column are dropped automatically, along with the column. ALTER TABLE COMPANY.EMPLOYEE DROP ADDRESS RESTRICT; Successful if no views or constraints reference the column. ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN DROP DEFAULT;

ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN SET DEFAULT ―333445555‖;

**BASIC QUERIES IN SQL**

 SQL has one basic statement for retrieving information from a database; the SLELECT statement

 This is *not the same as* the SELECT operation of the relational algebra

 Important distinction between SQL and the formal relational model;

 SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values

 Hence, an SQL relation (table) is a *multi-set* (sometimes called a bag) of tuples; it is *not* a set of tuples

 SQL relations can be constrained to be sets by using the CREATE UNIQUE INDEX command, or by using the DISTINCT option

 Basic form of the SQL SELECT statement is called a *mapping* of a *SELECT-FROM-WHERE block*

SELECT <attribute list> FROM <table list> WHERE <condition>

 <attribute list> is a list of attribute names whose values are to be retrieved by the query

 <table list > is a list of the relation names required to process the query

 <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query

**SIMPLE SQL QUERIES**

Basic SQL queries correspond to using the following operations of the relational algebra:

SELECT

PROJECT

JOIN

All subsequent examples uses COMPANY database as shown below:

**Example of a simple query on one relation**

**Retrieve the birth date and address of the employee whose name is 'John B. Smith'.**

SQL>SELECT BDATE, ADDRESS FROM EMPLOYEE WHERE FNAME='John' AND MINIT='B‘ AND LNAME='Smith‘

Similar to a SELECT-PROJECT pair of relational algebra operations: The SELECT-clause specifies the projection attributes and the WHERE-clause specifies the selection condition However, the result of the query may contain duplicate tuples







**Example of a simple query on two relations**

**Retrieve the name and address of all employees who work for the 'Research' department.**

SQL> SELECT FNAME, LNAME, ADDRESS FROM EMPLOYEE, DEPARTMENT WHERE DNAME='Research' AND DNUMBER=DNO

Similar to a SELECT-PROJECT-JOIN sequence of relational algebra operations (DNAME='Research') is a selection condition (corresponds to a SELECT operation in relational algebra) (DNUMBER=DNO) is a join condition (corresponds to a JOIN operation in relational algebra)

**Example of a simple query on three relations**

**For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.**

SQL> SELECT PNUMBER, DNUM, LNAME, BDATE, ADDRESS FROM PROJECT, DEPARTMENT, EMPLOYEE WHERE DNUM=DNUMBER AND MGRSSN=SSN AND PLOCATION='Stafford'

There are two join conditions The join condition DNUM=DNUMBER relates a project to its controlling department The join condition MGRSSN=SSN relates the controlling department to the employee who manages that department

**ALIASES, \* AND DISTINCT, EMPTY WHERE-CLAUSE**

 In SQL, we can use the same name for two (or more) attributes as long as the attributes are in different relations

 A query that refers to two or more attributes with the same name must qualify the attribute name with the relation name by prefixing the relation name to the attribute name

**Example:** EMPLOYEE.LNAME, DEPARTMENT.DNAME

 Some queries need to refer to the same relation twice. In this case, aliases are given to the relation name

**Example**

**For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.**

SQL> SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME FROM EMPLOYEE E S WHERE E.SUPERSSN=S.SSN

The alternate relation names E and S are called aliases or tuple variables for the EMPLOYEE relation We can think of E and S as two different copies of EMPLOYEE; E represents employees in role of supervisees and S represents employees in role of supervisors Aliasing can also be used in any SQL query for convenience. Can also use the AS keyword to specify aliases

SQL> SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME FROM EMPLOYEE AS E, EMPLOYEE AS S WHERE E.SUPERSSN=S.SSN

**UNSPECIFIED WHERE-clause**

A missing WHERE-clause indicates no condition; hence, all tuples of the relations in the FROM-clause are selected. This is equivalent to the condition WHERE TRUE

Example:

**Retrieve the SSN values for all employees.**

SQL> SELECT SSN FROM EMPLOYEE

If more than one relation is specified in the FROM-clause and there is no join condition, then the CARTESIAN PRODUCT of tuples is selected

Example:

SQL> SELECT SSN, DNAME FROM EMPLOYEE, DEPARTMENT

**Note:** It is extremely important not to overlook specifying any selection and join conditions in the WHERE-clause; otherwise, incorrect and very large relations may result

**USE OF \***

To retrieve all the attribute values of the selected tuples, a \* is used, which stands for all the attributes

Examples:

R**etrieve all the attribute values of EMPLOYEES who work in department 5.**

SQL> SELECT \* FROM EMPLOYEE WHERE DNO=5

**Retrieve all the attributes of an employee and attributes of DEPARTMENT he works in for every employee of ‘Research’ department.**

SQL> SELECT \* FROM EMPLOYEE, DEPARTMENT WHERE DNAME='Research' AND DNO=DNUMBER

**USE OF DISTINCT**

SQL does not treat a relation as a set; duplicate tuples can appear. To eliminate duplicate tuples in a query result, the keyword DISTINCT is used

Example: the result of **Q1c** may have duplicate SALARY values whereas **Q1d** does not have any duplicate values

SQL> SELECT SALARY FROM EMPLOYEE Q1d: SELECT **DISTINCT** SALARY FROM EMPLOYEE

**SET OPERATIONS**

SQL has directly incorporated some set operations such as union operation (UNION), set difference (MINUS) and intersection (INTERSECT) operations. The resulting relations of these set operations are sets of tuples; duplicate tuples are eliminated from the result. The set operations apply only to union compatible relations; the two relations must have the same attributes and the attributes must appear in the same order

**Make a list of all project numbers for projects that involve an employee whose last name is 'Smith' as a worker or as a manager of the department that controls the project.**

SQL> (SELECT PNAME FROM PROJECT, DEPARTMENT, EMPLOYEE WHERE DNUM=DNUMBER AND MGRSSN=SSN AND LNAME='Smith')

**UNION**

(SELECT PNAME FROM PROJECT, WORKS\_ON, EMPLOYEE WHERE PNUMBER=PNO AND ESSN=SSN AND NAME='Smith')

**NESTING OF QUERIES**

A complete SELECT query, called a nested query, can be specified within the WHERE-clause of another query, called the outer query. Many of the previous queries can be specified in an alternative form using nesting

**Retrieve the name and address of all employees who work for the 'Research' department.**

SQL: SELECT FNAME, LNAME, ADDRESS FROM EMPLOYEE WHERE DNO **IN** (SELECT DNUMBER FROM DEPARTMENT WHERE DNAME='Research' )

**Note:** The nested query selects the number of the 'Research' department. The outer query selects an EMPLOYEE tuple if its DNO value is in the result of either nested query. The comparison operator IN compares a value v with a set (or multi-set) of values V, and evaluates to TRUE if v is one of the elements in V

In general, we can have several levels of nested queries. A reference to an unqualified attribute refers to the relation declared in the innermost nested query. In this example, the nested query is not correlated with the outer query

**CORRELATED NESTED QUERIES**

If a condition in the WHERE-clause of a nested query references an attribute of a relation declared in the outer query, the two queries are said to be correlated. The result of a correlated nested query is different for each tuple (or combination of tuples) of the relation(s) the outer query

**Retrieve the name of each employee who has a dependent with the same first name as the employee**.

SQL> SELECT E.FNAME, E.LNAME FROM EMPLOYEE AS E WHERE E.SSN **IN** (SELECT ESSN FROM DEPENDENT WHERE ESSN=E.SSN AND E.FNAME=DEPENDENT\_NAME)

The nested query has a different result in the outer query. A query written with nested SELECT... FROM… WHERE... blocks and using the **= or IN** comparison operators can *always* be expressed as a single block query. For example, Q7 may be written as in Q7a

SQL> SELECT E.FNAME, E.LNAME FROM EMPLOYEE E, DEPENDENT D WHERE E.SSN=D.ESSN AND E.FNAME=D.DEPENDENT\_NAME

**THE EXISTS FUNCTION**

EXISTS is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. We can formulate Query 7 in an alternative form that uses EXIST.

SQL> SELECT FNAME, LNAME FROM EMPLOYEE WHERE **EXISTS** (SELECT \* FROM DEPENDENT WHERE SSN=ESSN AND FNAME=DEPENDENT\_NAME)

**Retrieve the names of employees who have no dependents***.*

SQL> SELECT FNAME, LNAME FROM EMPLOYEE WHERE **NOT EXISTS** (SELECT \* FROM DEPENDENT WHERE SSN=ESSN)

**Note:** In Q8, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If none exist, the EMPLOYEE tuple is selected

**EXPLICIT SETS**

It is also possible to use an explicit (enumerated) set of values in the WHERE-clause rather than a nested query

**Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.**

SQL> SELECT DISTINCT ESSN FROM WORKS\_ON WHERE PNO **IN (1, 2, 3) NULLS IN SQL QUERIES**

SQL allows queries that check if a value is NULL (missing or undefined or not applicable). SQL uses IS or IS NOT to compare NULLs because it considers each NULL value distinct from other NULL values, so equality comparison is not appropriate.

**Retrieve the names of all employees who do not have supervisors.**

SQL> SELECT FNAME, LNAME FROM EMPLOYEE WHERE SUPERSSN IS NULL

**Note:** If a join condition is specified, tuples with NULL values for the join attributes are not included in the result

**AGGREGATE FUNCTIONS**

Include COUNT, SUM, MAX, MIN, and AVG

**Find the maximum salary, the minimum salary, and the average salary among all employees.**

SQL> SELECT **MAX (SALARY), MIN(SALARY), AVG(SALARY)** FROM EMPLOYEE

**Note:** Some SQL implementations may not allow more than one function in the SELECT-clause

**Find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.**

SQL> SELECT **MAX (SALARY), MIN(SALARY), AVG(SALARY)** FROM EMPLOYEE, DEPARTMENT WHERE DNO=DNUMBER AND DNAME='Research'

**Queries 13 and 14: Retrieve the total number of employees in the company (Q13), and the number of employees in the 'Research' department (Q14).**

SQL> SELECT **COUNT (\*)** FROM EMPLOYEE

SQL> SELECT **COUNT (\*)** FROM EMPLOYEE, DEPARTMENT WHERE DNO=DNUMBER AND DNAME='Research‘

**GROUPING**

 In many cases, we want to apply the aggregate functions to subgroups of tuples in a relation

 Each subgroup of tuples consists of the set of tuples that have the same value for the grouping attribute(s)

 The function is applied to each subgroup independently

 SQL has a GROUP BY-clause for specifying the grouping attributes, which must also appear in the SELECT-clause

**For each department, retrieve the department number, the number of employees in the department, and their average salary.**

SQL> SELECT DNO, COUNT (\*), AVG (SALARY) FROM EMPLOYEE **GROUP BY** DNO

 In Q15, the EMPLOYEE tuples are divided into groups. Each group having the same value for the grouping attribute DNO

 The COUNT and AVG functions are applied to each such group of tuples separately

 The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples

 A join condition can be used in conjunction with grouping

**For each project, retrieve the project number, project name, and the number of employees who work on that project.**

SQL> SELECT PNUMBER, PNAME, COUNT (\*) FROM PROJECT, WORKS\_ON

WHERE PNUMBER=PNO GROUP BY PNUMBER, PNAME

**THE HAVING-CLAUSE**

Sometimes we want to retrieve the values of these functions for only those groups that satisfy certain conditions. The HAVING-clause is used for specifying a selection condition on groups (rather than on individual tuples)

**For each project on which more than two employees work, retrieve the project number, project name, and the number of employees who work on that project.**

SQL> SELECT PNUMBER, PNAME, COUNT (\*) FROM PROJECT, WORKS\_ON

WHERE PNUMBER=PNO GROUP BY PNUMBER, PNAME HAVING COUNT (\*) > 2

**SUBSTRING COMPARISON**

The LIKE comparison operator is used to compare partial strings. Two reserved characters are used: **'%'** (or '\*' in some implementations) replaces an arbitrary number of characters, and **'\_'** replaces a single arbitrary character.

**Retrieve all employees whose address is in Houston, Texas. Here, the value of the ADDRESS attribute must contain the substring 'Houston,TX‘ in it.**

SQL> SELECT FNAME, LNAME FROM EMPLOYEE WHERE ADDRESS LIKE '%Houston,TX%'

**Retrieve all employees who were born during the 1950s.**

Here, '5' must be the 8th character of the string (according to our format for date), so the BDATE value is '\_\_\_\_\_\_\_5\_', with each underscore as a place holder for a single arbitrary character.

SQL> SELECT FNAME, LNAME FROM EMPLOYEE WHERE BDATE **LIKE** '**\_\_\_\_\_\_\_**5**\_**‘

**Note:** The LIKE operator allows us to get around the fact that each value is considered atomic and indivisible. Hence, in SQL, character string attribute values are not atomic

**ORDER BY**

The ORDER BY clause is used to sort the tuples in a query result based on the values of some attribute(s)

**Retrieve a list of employees and the projects each works in, ordered by the employee's department, and within each department ordered alphabetically by employee last name.**

SQL> SELECT DNAME, LNAME, FNAME, PNAME FROM DEPARTMENT, EMPLOYEE, WORKS\_ON, PROJECT WHERE DNUMBER=DNO AND SSN=ESSN

AND PNO=PNUMBER ORDER BY DNAME, LNAME

The default order is in ascending order of values. We can specify the keyword DESC if we want a descending order; the keyword ASC can be used to explicitly specify ascending order, even though it is the default

Ex: ORDER BY DNAME **DESC**, LNAME **ASC**, FNAME **ASC**

**MORE EXAMPLE QUERIES:**

**Retrieve the names of all employees who have two or more dependents.**

SQL> SELECT LNAME, FNAME FROM EMPLOYEE WHERE (SELECT COUNT (\*) FROM DEPENDENT WHERE SSN=ESSN) ≥ 2);

**List the names of managers who have least one dependent.**

SQL> SELECT FNAME, LNAME FROM EMPLOYEE WHERE EXISTS (SELECT \* FROM DEPENDENT WHERE SSN=ESSN) AND EXISTS ( SELECT \* FROM DEPARTMENT WHERE SSN=MGRSSN );

**SPECIFYING UPDATES IN SQL**

There are three SQL commands to modify the database: **INSERT**, **DELETE**, and **UPDATE.**

**INSERT**

 In its simplest form, it is used to add one or more tuples to a relation

 Attribute values should be listed in the same order as the attributes were specified in the **CREATE TABLE** command

**Example:**

SQL> INSERT INTO EMPLOYEE VALUES ('Richard','K','Marini', '653298653', '30-DEC-52', '98 Oak Forest,Katy,TX', 'M', 37000,'987654321', 4 )

 An alternate form of INSERT specifies explicitly the attribute names that correspond to the values in the new tuple. Attributes with NULL values can be left out

**Example:** Insert a tuple for a new EMPLOYEE for whom we only know the FNAME, LNAME, and SSN attributes.

SQL> INSERT INTO EMPLOYEE (FNAME, LNAME, SSN)VALUES ('Richard', 'Marini', '653298653')

**Important Note**: Only the constraints specified in the DDL commands are automatically enforced by the DBMS when updates are applied to the database. Another variation of INSERT allows insertion of multiple tuples resulting from a **query** into a relation

**Example:** Suppose we want to create a temporary table that has the name, number of employees, and total salaries for each department. A table DEPTS\_INFO is created first, and is loaded with the summary information retrieved from the database by the query.

CREATE TABLE DEPTS\_INFO

(DEPT\_NAME VARCHAR (10), NO\_OF\_EMPS INTEGER, TOTAL\_SAL INTEGER);

INSERT INTO DEPTS\_INFO (DEPT\_NAME, NO\_OF\_EMPS, TOTAL\_SAL) SELECT DNAME, COUNT (\*), SUM (SALARY) FROM DEPARTMENT, EMPLOYEE WHERE DNUMBER=DNO GROUP BY DNAME ;

**Note:** The DEPTS\_INFO table may not be up-to-date if we change the tuples in either the DEPARTMENT or the EMPLOYEE relations *after* issuing the above. We have to create a view (see later) to keep such a table up to date.

**DELETE**

 Removes tuples from a relation. Includes a WHERE-clause to select the tuples to be deleted

 Referential integrity should be enforced

 Tuples are deleted from only *one table* at a time (unless CASCADE is specified on a referential integrity constraint)

 A missing WHERE-clause specifies that *all tuples* in the relation are to be deleted; the table then becomes an empty table

 The number of tuples deleted depends on the number of tuples in the relation that satisfy the WHERE-clause

Examples:

1: DELETE FROM EMPLOYEE WHERE LNAME='Brown‘;

2: DELETE FROM EMPLOYEE WHERE SSN='123456789‘;

3: DELETE FROM EMPLOYEE WHERE DNO IN (SELECT DNUMBER FROM DEPARTMENT WHERE DNAME='Research');

4: DELETE FROM EMPLOYEE;

**UPDATE**

 Used to modify attribute values of one or more selected tuples

 A WHERE-clause selects the tuples to be modified

 An additional SET-clause specifies the attributes to be modified and their new values

 Each command modifies tuples *in the same relation*

 Referential integrity should be enforced

**Example1:** Change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

SQL> UPDATE PROJECT SET PLOCATION = 'Bellaire', DNUM = 5 WHERE PNUMBER=10;

**Example2:** Give all employees in the 'Research' department a 10% raise in salary.

SQL> UPDATE EMPLOYEE SET SALARY = SALARY \*1.1 WHERE DNO IN (SELECT DNUMBER FROM DEPARTMENT WHERE DNAME='Research');

**VIEWS IN SQL**

 A view is a single *virtual table* that is derived from other tables. The other tables could be base tables or previously defined view.

 Allows for limited update operations Since the table may not physically be stored

 Allows full query operations

 A convenience for expressing certain operations

 A view does not necessarily exist in physical form, which limits the possible update operations that can be applied to views.

In Oracle, view is a virtual table that does not physically exist. It is stored in Oracle data dictionary and do not store any data. It can be executed when called.

A view is created by a query joining one or more tables.

**Oracle CREATE VIEW**

**Syntax:**

**CREATE** **VIEW** view\_name **AS** **SELECT** columns **FROM** tables **WHERE** conditions;

**Parameters:**

**view\_name:** It specifies the name of the Oracle VIEW that you want to create.

**Example:**

Let's take an example to create view. In this example, we are creating two tables suppliers and orders first.

**Suppliers table:**

**CREATE** **TABLE**  "SUPPLIERS" ("SUPPLIER\_ID" NUMBER, "SUPPLIER\_NAME"

VARCHAR2(4000), "SUPPLIER\_ADDRESS" VARCHAR2(4000));

# **PL/SQL**

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL. PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java. This tutorial will give you great understanding on PL/SQL to proceed with Oracle database and other advanced RDBMS concepts.

Following are certain notable facts about PL/SQL −

* PL/SQL is a completely portable, high-performance transaction-processing language.
* PL/SQL provides a built-in, interpreted and OS independent programming environment.
* PL/SQL can also directly be called from the command-line **SQL\*Plus interface**.
* Direct call can also be made from external programming language calls to database.
* PL/SQL's general syntax is based on that of ADA and Pascal programming language.
* Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

## Features of PL/SQL

PL/SQL has the following features −

* PL/SQL is tightly integrated with SQL.
* It offers extensive error checking.
* It offers numerous data types.
* It offers a variety of programming structures.
* It supports structured programming through functions and procedures.
* It supports object-oriented programming.
* It supports the development of web applications and server pages.

## Advantages of PL/SQL

PL/SQL has the following advantages −

* SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
* PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
* PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
* PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
* Applications written in PL/SQL are fully portable.
* PL/SQL provides high security level.
* PL/SQL provides access to predefined SQL packages.
* PL/SQL provides support for Object-Oriented Programming.
* PL/SQL provides support for developing Web Applications and Server Pages.

In this chapter, we will discuss the Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts –

**Declarations**

This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

**Executable Commands**

This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.

**Exception Handling**

This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block −DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling>

END;

Exampl:

DECLARE

message varchar2(20):= 'Hello, World!';

BEGIN

dbms\_output.put\_line(message);

END;

/

**SQL TRIGGERS**

 Objective: to monitor a database and take initiate action when a condition occurs

 Triggers are nothing but the procedures/functions that involve actions and fired/executed automatically whenever an event occurs such as an insert, delete, or update operation or pressing a button or when mouse button is clicked.

In this chapter, we will discuss Triggers in PL/SQL. Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

* A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
* A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
* A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

### **Benefits of Triggers**

Triggers can be written for the following purposes −

* Generating some derived column values automatically
* Enforcing referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

## Creating Triggers

The syntax for creating a trigger is −

CREATE [OR REPLACE ] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col\_name]

ON table\_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Where,

* CREATE [OR REPLACE] TRIGGER trigger\_name − Creates or replaces an existing trigger with the *trigger\_name*.
* {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
* {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
* [OF col\_name] − This specifies the column name that will be updated.
* [ON table\_name] − This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
* [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
* WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

### **Example**

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

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 |

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

The following program creates a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

CREATE OR REPLACE TRIGGER display\_salary\_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW

WHEN (NEW.ID > 0)

DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.salary - :OLD.salary;

dbms\_output.put\_line('Old salary: ' || :OLD.salary);

dbms\_output.put\_line('New salary: ' || :NEW.salary);

dbms\_output.put\_line('Salary difference: ' || sal\_diff);

END;

/

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

Trigger created.

The following points need to be considered here −

OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.

If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

**PL/SQL - Procedures**

In this chapter, we will discuss Procedures in PL/SQL. A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created −

At the schema level

Inside a package

Inside a PL/SQL block

At the schema level, subprogram is a standalone subprogram. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a packaged subprogram. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

**Functions** − These subprograms return a single value; mainly used to compute and return a value.

**Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a PL/SQL procedure. We will discuss PL/SQL function in the next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts –

**Declarative Part**

It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

**Executable Part**

This is a mandatory part and contains statements that perform the designated action.

**Exception-handling**

This is again an optional part. It contains the code that handles run-time errors.

**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;

# **PL/SQL - Functions**

In this chapter, we will discuss the functions in PL/SQL. A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

## 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 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.

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

n! = n\*(n-1)!

= n\*(n-1)\*(n-2)!

...

= n\*(n-1)\*(n-2)\*(n-3)... 1

The following program calculates the factorial of a given number by calling itself recursively −

DECLARE

num number;

factorial number;

FUNCTION fact(x number)

RETURN number

IS

f number;

BEGIN

IF x=0 THEN

f := 1;

ELSE

f := x \* fact(x-1);

END IF;

RETURN f;

END;

BEGIN

num:= 6;

factorial := fact(num);

dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial);

END;

/

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

Factorial 6 is 720

PL/SQL procedure successfully completed.

**Task1**

**Conceptual Design through FTR**

Aim:

Using basic database design methodology and ER modeler, design Entity Relationship Diagram by satisfying the following sub tasks:

* 1. **a** Identifying the entities.

1. **b** Identifying the attributes.

**1. c** Identification of relationships, cardinality, type of relationship.

**1. d** Reframing the relations with keys and constraint.

1. **e** Using create, develop ER/ER diagram

**1.a Identifying the entities**

1.a.1 CricketBoard

1.a.2 Team

1.a.3 Player

1.a.4 Match

1.a.5 Ground

1.a.6 Umpire

**1.b Identifyingtheattributes**

1.b.1 CricketBoard(BoardID, Name, Address, Contact\_No)

1.b.2 Team(TeamID, Name, Coach, Captain)

1.b.3 Player(PlayerID, FName, LName, Age, DateofBirth, PlayingRole, email, contact\_no)

1.b.4 Match( MatchID, Date, Time, Result)

1.b.5 Ground(GroundID, Name, Location, Capacity)

# 1.b.6 Umpire(UmpireID, FName, LName, Age, DateofBirth, Country, email, contact\_no)

# 1.c Identification of relationships, cardinality, type of relationship.

1.c.1 **Board-Team Relationship:** The Board will have a **one-to-many** relationship with Teams since the board can have multiple teams affiliated with it, but a team can only be associated with one board.

1.c.2 **Team-Player Relationship:** Teams and Players will have a **many-to-many** relationship since a team can have multiple players, and a player can be a part of multiple teams over time.

1.c,3. **Match-Team Relationship:** Matches will have a **many-to-many** relationship with Teams, as a match involves two teams, and a team can participate in multiple matches.

1.c.4. **Match-Ground Relationship:** Matches will have a **one-to-one** relationship with Grounds, as each match takes place in one specific ground.

**1.d Reframing the relations with keys and constraint**

**1.d.1 Create Table CricketBoard:**

**SQL>create table CricketBoard(**BoardID varchar(10) PRIMARY KEY, Name varchar(30), Address varchar(50), Contact\_No number);

Table Created

SQL>DESC CricketBoard

|  |  |  |
| --- | --- | --- |
| Column | NULL | TYPE |
| BoardID | NOT NULL | VARCHAR(10) |
| Name | - | VARCHAR(30) |
| Address | - | VARCHAR(50) |
| Contact\_No | - | NUMBER |

**1.d.2 Create Table Team:**

**SQL> create table Team(TeamID varchar(6) PRIMARY KEY, BoardID varchar(10), Name varchar(30), Coach varchar(30), Captain varchar(30), FOREIGN KEY(BoardID) REFERENCES CricketBoard(BoardID));**

**Table created.**

**SQL> DESC TEAM**

Name Null? Type

----------------------------------------- -------- ----------------------------

TEAMID NOT NULL VARCHAR2(6)

BOARDID NOT NULL VARCHAR2(10)

NAME - VARCHAR2(30)

COACH - VARCHAR2(30)

CAPTAIN - VARCHAR2(30)

**1.d.3 Create Table Player:**

SQL> CREATE table Player(PlayerID varchar(6) PRIMARY KEY, TeamID varchar(6), FName varchar(30), LName varchar(30), Age number(5,2), DateofBirth date, PlayingRole varchar(25), email varchar(40), contact\_no number, FOREIGN KEY(TeamID) REFERENCES Team(TeamID));

Table created.

SQL> DESC PLAYER

Name Null? Type

----------------------------------------- -------- ----------------------------

PLAYERID NOT NULL VARCHAR2(6)

TEAMID NOT NULL VARCHAR2(6)

FNAME VARCHAR2(30)

LNAME VARCHAR2(30)

AGE NUMBER(5,2)

DATEOFBIRTH DATE

PLAYINGROLE VARCHAR2(25)

EMAIL VARCHAR2(40)

CONTACT\_NO NUMBER

**1.d.4 Create Table Match:**

SQL> create table Match( MatchID varchar(10), TeamID1 varchar(6), TeamID2 varchar(6), PlayerID varchar(6), Match\_Date date, Time1 number, Result varchar(20), PRIMARY KEY(MatchID,PlayerID), FOREIGN KEY(TeamID1) REFERENCES team(TeamID), FOREIGN KEY(TeamID2) REFERENCES team(TeamID), FOREIGN KEY(PlayerID) REFERENCES Player(PLayerID));

Table created.

SQL> DESC Match

Name Null? Type

----------------------------------------- -------- ----------------------------

MATCHID NOT NULL VARCHAR2(10)

TEAMID1 NOT NULL VARCHAR2(6)

TEAMID2 NOT NULL VARCHAR2(6)

PLAYERID NOT NULL VARCHAR2(6)

MATCH\_DATE DATE

TIME1 NUMBER

RESULT VARCHAR2(20)

**1.d.5 Create Table Ground:**

SQL> create table Ground(GroundID varchar(10) PRIMARY KEY, MatchID Varchar(10), Name varchar(30), Location varchar(30), Capacity number, FOREIGN KEY(MatchID) REFERENCES Match(MatchID));

Table created.

SQL> DESC Ground

Name Null? Type

----------------------------------------- -------- ----------------------------

GROUNDID NOT NULL VARCHAR2(10)

MATCHID NOT NULL VARCHAR2(10)

NAME VARCHAR2(30)

LOCATION VARCHAR2(30)

CAPACITY NUMBER

**1.d.6 Create Table Umpire:**

**SQL> Create Table Umpire(UmpireID varchar(10) PRIMARY KEY, FName varchar(30), LName varchar(30), Age number(5,2), DateofBirth date, Country varchar(30), email varchar(40), contact\_no number);**

SQL> DESC Umpire

Name Null? Type

----------------------------------------- -------- ----------------------------

UMPIREID NOT NULL VARCHAR2(10)

FNAME VARCHAR2(30)

LNAME VARCHAR2(30)

AGE NUMBER(5,2)

DATEOFBIRTH DATE

COUNTRY VARCHAR2(30)

EMAIL VARCHAR2(40)

CONTACT\_NO NUMBER

**1.d.6 Create Table Umpire\_Umpired:**

SQL> create table Umpire\_Umpired(UmpireID varchar(10), MatchID Varchar(10), GroundID varchar(10), FOREIGN KEY(UmpireID) REFERENCES Umpire(UmpireID), FOREIGN KEY(MatchID) REFERENCES Match(MatchID), FOREIGN KEY(GroundID) REFERENCES Ground(GroundID));

Table created.

SQL> DESC Umpire

Name Null? Type

----------------------------------------- -------- ----------------------------

UMPIREID NOT NULL VARCHAR2(10)

GROUNDID NOT NULL VARCHAR2(10)

MATCHID NOT NULL VARCHAR2(10)

**Result:**

Thus the database design methodology and ER Model design diagram has been completed successfully.

**TASK2**

**Generating Design of other traditional database model**

**Aim:**

Creating Hierarchical/Network model of the database by enhancing the sound abstract data by performing following tasks using forms of inheritance:

2. a Identify the specificity of each relationship, find and form surplus relations.

2. b Check is-a hierarchy/has-a hierarchy and performs generalization and/or specialization relationship.

2. c Find the domain of the attribute and perform check constraint to the applicable.

2. d Rename the relations.

2. e Perform SQL Relations using DDL, DCL commands.

**2. a Identify the specificity of each relationship, find and form surplus relations.**

**Relationship: Cricket Board manages Team (one-to-many)**

Specificity: One Cricket Board manages one or more Teams, but each Team is managed by only one Cricket Board.

Surplus Relation: No surplus relation is needed for this relationship since it is already one-to-many.

**Relationship: Team has Player (many-to-one)**

Specificity: One Team can have many Players, but each Player belongs to only one Team.

Surplus Relation: No surplus relation is needed for this relationship since it is already many-to-one.

**Relationship: Match involves Team (many-to-many)**

Specificity: One Match involves two Teams, and each Team can participate in multiple Matches.

Surplus Relation: No surplus relation is needed for this relationship since it is already many-to-many.

**Relationship: Match has Umpire (many-to-many)**

Specificity: One Match can have multiple Umpires, and each Umpire can officiate multiple Matches.

Surplus Relation: No surplus relation is needed for this relationship since it is already many-to-many.

Based on the specificity analysis, all the relationships in the ER diagram are appropriately represented, and there are no surplus relations required for this particular model. Each relationship reflects the correct cardinality and participation constraints as per the description provided earlier.

**2.b Check is-a hierarchy/has -a hierarchy and performs generalization and/or specialization relationship.**

Generalization

In the ER diagram for the Tamil Nadu Cricket Board (TNCA) described earlier, we can identify potential generalizations based on common attributes or relationships among entities. Here's an example of a possible generalization:

**Entities:**

Player

Umpire

**Attributes:**

The above entities have common attributes like First\_Name, Last\_Name, Date\_of\_Birth, age, Contact\_No, and Email.

**Potential Generalization:**

Create a superclass called "Person" to represent the common attributes shared by Player and Umpire. The "Person" entity would have the following attributes:

Person\_ID (primary key)

First\_Name

Last\_Name

Date\_of\_Birth

Age

Contact\_Number

Email

**Subclasses:**

Player: Inherited attributes from "Person" and add specific attributes like Player\_ID.

Umpire: Inherited attributes from "Person" and add specific attributes like Umpire\_ID.

Umpire

Umpire\_ID, FName, LName, Age, DateofBirth, email, contact\_no

Player

Player\_ID, FName, LName, Age, DateofBirth, email, contact\_no

Person

Person\_ID,

FName, LName, Age, DateofBirth, email, contact\_no,

Role

By using generalization, we can reduce data redundancy, improve data integrity, and simplify the structure of the ER diagram. This approach also allows for easier maintenance and updates, as changes made to the attributes shared by all "Person" entities will be automatically reflected in the subclasses.

**Specialization**

In the context of Entity-Relationship (ER) diagrams, specialization refers to the process of defining subtypes within an entity type. It allows, to represent entities that have specific attributes or relationships distinct from the general attributes or relationships of the parent entity.

In the case of the Tamil Nadu Cricket Board Association, let's consider the specialization of the "Player" entity into two subtypes: "Batsman" and "Bowler." This specialization is based on the specific roles that players can have in cricket.

Here's the modified ER diagram with the specialization:

Player

PlayerID, FName, LName, Age, DateofBirth, email, contact\_no

Bowler

Bowler\_ID

Bowling\_avg

Batsman

Batsman\_ID

Batting\_avg

**2. c Find the domain of the attribute and perform check constraint to the applicable.**

For the purpose of illustration, I'll assume we are considering the "age" attribute of the "Player" entity from the ER diagram of the Tamil Nadu Cricket Association.

Finding the domain of the "age" attribute:

The "age" attribute typically represents the age of a player, and it should be a positive integer or a non-negative integer depending on how you handle the birth dates of players. For the sake of simplicity, let's assume it's a positive integer.

Check constraint to enforce the domain:

To enforce the domain on the "age" attribute and ensure that only valid values are allowed, we can create a check constraint in the database schema. The check constraint will specify the condition that the "age" attribute must satisfy.

Suppose your database schema language is SQL, here's an example of how you can add the check constraint:

**SQL> ALTER TABLE Player ADD CONSTRAINT check\_con CHECK (age>= 20);**

**Table altered.**

**2.d** Rename the relations:

Renaming a table (relation) in SQL can be accomplished using the ALTER TABLE statement with the RENAME TO clause. The specific syntax for renaming tables varies slightly between different database management systems.

Here's the syntax for renaming a column in the Table:

SQL> Alter table Umpire RENAME column contact\_no TO phone\_no;

Table altered.

SQL> DESC Umpire

Name Null? Type

----------------------------------------- -------- ----------------------------

UMPIREID VARCHAR2(10)

FNAME VARCHAR2(30)

LNAME VARCHAR2(30)

AGE NUMBER(5,2)

DATEOFBIRTH DATE

COUNTRY VARCHAR2(30)

EMAIL VARCHAR2(40)

PHONE\_NO NUMBER

**2.e** Perform SQL Relations using DDL, DCL commands.

DCL stands for "Data Control Language," which is a subset of SQL (Structured Query Language) used to control access to data in a database. DCL commands are responsible for managing user permissions, granting privileges, and controlling data security within a database system. There are two primary DCL commands:

1. Grant
2. Revoke

**GRANT:**

The GRANT command is used to provide specific privileges to users or roles, allowing them to perform certain actions on database objects (e.g., tables, views, procedures). Privileges may include SELECT, INSERT, UPDATE, DELETE, EXECUTE, and more.

**SQL> create user Raj identified by kumar;**

User created.

SQL> grant resource to raj;

Grant succeeded.

SQL> grant create session to raj;

Grant succeeded.

SQL> conn

Enter user-name: raj

Enter password:

Connected.

SQL> create table emp(eno number,ename varchar(10));

Table created.

SQL> conn system/manager

Connected.

SQL> grant all on Umpire to Raj;

Grant succeeded.

**Result:**

Thus the Hierarchical model and Network model has been successfully created.

**TASK 3**

**Using Clauses, Operators and Functions in queries**

Aim:

To perform the query processing on databases for different retrieval results of queries using DML, DRL operations using aggregate, date, string, indent functions, set clauses and operators.

* To retrieve the location of a particular match conducted by its MatchID
* To retrieve the Players detail whose name start with ‘A’.
* Add a column Batting and Bowling in Player table.
* To count the number of right-hand batsman in a team.
* To display the CricketBoard details for the BoardIDs 'BID01', 'BID03', and 'BID06'.
* To select the names and IDs of players who are left-hand bowlers.
* To find the UmpireID of umpires who have not umpired any match.

**CricketBoard:**

|  |  |  |  |
| --- | --- | --- | --- |
| **BoardID** | **Name** | **Address** | **Contact\_No** |
| BID01 | Chennai Cricket Board | Chennai | 9988776699 |
| BID02 | Tiruvallur Cricket Board | Chennai | 9977886699 |
| BID03 | Viluppuram Cricket Board | Viluppuram | 9966886699 |
| BID04 | Trichy Cricket Board | Trichy | 9955886699 |
| BID05 | Madurai Cricket Board | Madurai | 9944886699 |
| BID06 | Tuticorin Cricket Board | Tuticorin | 9933886699 |
| BID07 | Selam Cricket Board | Selam | 9922886699 |
| BID08 | Tiruppur Cricket Board | Tiruppur | 9911886699 |

**Team:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TeamID** | **BoardID** | **Name** | **Coach** | **Captain** |
| CCB01 | BID01 | ABS EXPRESS | G.D.RAMESH | SAMPATH KUMAR |
| CCB02 | BID01 | AVG EXPRESS | T.KARTHIK | Y.JOHN |
| TCB01 | BID02 | ANGRY BARD | TOM BABU | CINIL JOHN |
| TCB02 | BID02 | TIGER ROCK | S.KANNAN | BEN GEORGE |
| TRICB01 | BID04 | ROCK | K.PAUL | K.MUTHU |
| VCB01 | BID03 | RAINBOW | S.RAJESHKUMAR | MANIMARAN |
| MCB01 | BID05 | PANTHER | SARAVANAN | R.SUNILKUMAR |
| TUCB01 | BID06 | THUNDER | D ALEX | BARATHI |
| SCB01 | BID07 | EAGLE | SOMU | SRI HARI |
| TICB01 | BID08 | KINGS | D ANAND | MATHAN |

**Player:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **PlayerID** | **TeamID** | **FName** | **LName** | **Age** | **DateofBirth** | **PlayingRole** | **email** | **contact\_no** | **Batting** | **Bowling** |
| 1 | CCB01 | Raj | N | 27 | 29-JUN-1996 | Bowler | rajn@gmail.com | 9191910101 | null | left-hand |
| 33 | CCB01 | Balaji | D | 23 | 02-JAN-1999 | Batsman | balajid@gmail.com | 9191910031 | right-hand | *Null* |
| 02 | CCB02 | Krishna | R | 23 | 02-JAN-1999 | Bowler | krishnar@gmail.com | 9191930103 | *null* | right-hand |
| 18 | CCB02 | Kishore | K | 24 | 02-SEP-1998 | ALL ROUNDER | kishorek@gmail.com | 9291930105 | left-hand | left-hand |
| 19 | TCB01 | Karthick | K | 24 | 14-SEP-1998 | Batsman | karthickk@gmail.com | 9292930107 | right-hand | *Null* |
| 62 | TCB01 | Amar | J | 22 | 21-SEP-1998 | Batsman | Amarj@gmail.com | 9292930508 | right-hand | *Null* |
| 102 | TCB02 | Akash | G | 21 | 26-SEP-1999 | Batsman | Amarj@gmail.com | 9292930510 | right-hand | *Null* |
| 12 | TCB02 | Premkumar | S | 21 | 13-OCT-1999 | Bowler | Premkumars@gmail.com | 9592930517 | *null* | right-hand |
| 01 | VCB02 | Prem | V | 23 | 13-APR-1997 | Bowler | Premkumars@gmail.com | 9592950517 | *null* | left-hand |
| 21 | VCB02 | Kali | J | 21 | 11-APR-2002 | Batsman | Kalij@gmail.com | 9592950630 | right-hand | *Null* |
| 61 | VCB01 | Kamalesh | A | 22 | 21-JUN-2001 | Batsman | Kamalesha@gmail.com | 9592958730 | right-hand | *Null* |
| 66 | VCB01 | Ganesh | V | 24 | 21-JUN-1998 | Batsman | Ganeshv@gmail.com | 9592958790 | right-hand | *Null* |
| 303 | TRICB01 | Arun | T | 24 | 21-OCT-1998 | Batsman | Ganeshv@gmail.com | 9592958450 | right-hand | *Null* |
| 313 | TRICB01 | Srinivasan | N | 24 | 21-OCT-1998 | Batsman | srinivasann@gmail.com | 9992958450 | right-hand | *Null* |

**Match:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **MatchID** | **TeamID** | **TeamID** | **Match\_Date** | **Time1** | **Result** |
| M01 | CCB01 | TCB01 | 22-JUN-2022 | 1.3 | TCB01 - WIN |
| M02 | CCB02 | TCB02 | 22-JUN-2022 | 8.3 | CCB01 - WIN |
| M03 | TRIBCB02 | TCB01 | 24-JUN-2022 | 8.3 | TCB01 - WIN |
| M04 | TRIBCB01 | TCB02 | 25-JUN-2022 | 8.3 | TRICB01 - WIN |

**Ground:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **GroundID** | **MatchID** | **Name** | **Location** | **Capacity** |
| GID01 | M01 | Nehru | Chennai | 10000 |
| GID02 | M02 | GK | Coimbatore | 10000 |
| GID03 | M03 | Sankar | Nellai | 6000 |

**Umpire:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **UmpireID** | **FName** | **LName** | **Age** | **DateofBirth** | **Country** | **email** | **contact\_no** |
| UID01 | Venkatesh | T | 45 | 21-JUN-1978 | INDIA | venkatesh@gmail.com | 9665571435 |
| UID02 | Muthukumar | R | 46 | 01-JUN-1979 | INDIA | mutukumarr@gmail.com | 9665571460 |
| UID03 | Somu | K | 42 | 01-JUN-1983 | INDIA | somuk@gmail.com | 9664471460 |

**Umpire\_Umpired:**

|  |  |  |
| --- | --- | --- |
| **UmpireID** | **MatchID** | **GroundID** |
| UID01 | M01 | GID01 |
| UID02 | M03 | GID02 |

3.1: To retrieve the location of a particular match conducted by its MatchID

SQL> SELECT LOCATION FROM GROUND WHERE MatchID=’M03’;

Resul:

|  |
| --- |
| **Location** |
| Nellai |

3.2: To retrieve the Players detail whose name start with ‘A’.

SQL> Select \* from Player where FName like 'A%';

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **PlayerID** | **TeamID** | **FName** | **LName** | **Age** | **DateofBirth** | **PlayingRole** | **Email** | **contact\_no** |
| 62 | TCB01 | Amar | J | 22 | 21-SEP-1998 | Batsman | [Amarj@gmail.com](mailto:Amarj@gmail.com) | 9292930508 |
| 102 | TCB02 | Akash | G | 21 | 26-SEP-1999 | Batsman | [Amarj@gmail.com](mailto:Amarj@gmail.com) | 9292930510 |
| 303 | TRICB01 | Arun | T | 24 | 21-OCT-1998 | Batsman | [Ganeshv@gmail.com](mailto:Ganeshv@gmail.com) | 9592958450 |

3.3: Add a column Batting and Bowling in Player table.

SQL> Alter table player add Batting varchar(10);

Table Altered

SQL> Alter table player add Bowling varchar(10);

Table Altered

3.4: To count the number of right-hand batsman in a team.

SQL> Select count(\*) from Player where Batting='right-hand';

Result:

|  |
| --- |
| **count(\*)** |
| 9 |

3.5: To display the CricketBoard details for the BoardIDs 'BID01', 'BID03', and 'BID06'.

SQL> Select \* from CricketBoard where BoardID in('BID01','BID03','BID06');

|  |  |  |  |
| --- | --- | --- | --- |
| **BoardID** | **Name** | **Address** | **Contact\_No** |
| BID01 | Chennai Cricket Board | Chennai | 9988776699 |
| BID03 | Viluppuram Cricket Board | Viluppuram | 9966886699 |
| BID06 | Tuticorin Cricket Board | Tuticorin | 9933886699 |

3.6: To select the names and IDs of players who are left-hand bowlers.

SQL> Select playerID, FName, LName from player where Bowling='left-hand';

|  |  |  |
| --- | --- | --- |
| **PlayerID** | **FName** | **LName** |
| 1 | Raj | N |
| 18 | Kishore | K |
| 01 | Prem | V |

3.7: To find the UmpireID of umpires who have not umpired any match.

SQL> select a.UmpireID from Umpire a where UmpireID NOT IN(select UmpireID from Umpire\_Umpired);

Result:

|  |
| --- |
| **UmpireID** |
| UID03 |

**Result:**

Thus the query processing on database for different retrieval result of query using Clauses, Operators and Functions in queries has been performed successfully.

**TASK 4**

**Using Functions in Queries and Writing Sub Queries**

Aim:

To perform the advanced query processing and test its heuristics using designing of optimal correlated and nested sub queries such as finding summary statistics.

1. To retrieve all team details, including the count of winning matches for each team
2. To retrieve the total number of 'Tie' matches in a team-wise manner.
3. To retrieve the team details who won the matches.
4. To retrieve players and match details of players who are above 25 years old.
5. To retrieve the details of Team who have not played any matches.
6. To retrieve the teamid, boardid, teamname, and playername for a particular playerid given.
   1. To retrieve all team details, including the count of winning matches for each team.

SQL> SELECT t.TeamID, t.Name AS TeamName, t.Coach, t.Captain, COUNT(m.MatchID) AS WinningMatchCount FROM Team t LEFT JOIN Match m ON t.TeamID = substr(m.result,1,5) GROUP BY t.TeamID, t.Name, t.Coach, t.Captain;

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TeamID** | **TeamName** | **Coach** | **Captain** | **WinningMatchCount** |
| CCB01 | ABS EXPRESS | G.D.RAMESH | SAMPATH KUMAR | 1 |
| CCB02 | AVG EXPRESS | T.KARTHIK | Y.JOHN | 0 |
| MCB01 | PANTHER | SARAVANAN | R.SUNILKUMAR | 0 |
| SCB01 | EAGLE | SOMU | SRI HARI | 0 |
| TCB01 | ANGRY BARD | TOM BABU | CINIL JOHN | 2 |
| TCB02 | TIGER ROCK | S.KANNAN | BEN GEORGE | 0 |
| TICB01 | KINGS | D ANAND | MATHAN | 0 |
| TRICB01 | ROCK | K.PAUL | K.MUTHU | 0 |
| TUCB01 | THUNDER | D ALEX | BARATHI | 0 |
| VCB01 | RAINBOW | S.RAJESHKUMAR | MANIMARAN | 0 |

4.2 To retrieve the total number of 'Tie' matches in a team-wise manner.

SQL> SELECT t.Name AS TeamName, COUNT(\*) AS TotalTieMatches FROM Team t JOIN Match\_result mt ON t.TeamID = mt.TeamID JOIN Match\_result m ON mt.MatchID = m.MatchID WHERE m.Result = 'Tie' GROUP BY t.Name;

|  |  |
| --- | --- |
| **TeamName** | **TotalTieMatches** |
| ROCK | 1 |

* 1. To retrieve the team details who won the matches.

SQL> select \* from team where teamID in (select mr.teamID from match\_result mr left join team t on mr.teamId=t.teamID where mr.result='Win');

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TeamID** | **BoardID** | **Name** | **Coach** | **Captain** |
| CCB01 | BID01 | ABS EXPRESS | G.D.RAMESH | SAMPATH KUMAR |
| TCB01 | BID02 | ANGRY BARD | TOM BABU | CINIL JOHN |
| TRICB01 | BID04 | ROCK | K.PAUL | K.MUTHU |

* 1. To retrieve players and match details of players who are above 25 years old.

SQL> SELECT p.PlayerID, p.FName AS PlayerName, p.Age, m.MatchID, m.match\_Date, m.Time1, m.Result FROM Player p, match m where p.playerID in(select playerID from player where age>25);

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PlayerID** | **PlayerName** | **Age** | **MatchID** | **Match\_Date** | **Time1** | **Result** |
| 1 | Raj | 27 | M01 | 22-JUN-2022 | 1.3 | TCB01 - WIN |
| 1 | Raj | 27 | M02 | 22-JUN-2022 | 8.3 | CCB01 - WIN |
| 1 | Raj | 27 | M03 | 24-JUN-2022 | 8.3 | TCB01 - WIN |
| 1 | Raj | 27 | M04 | 25-JUN-2022 | 8.3 | TRICB01 - WIN |
| 1 | Raj | 27 | M05 | 04-APR-2023 | 7.3 | Tie |

* 1. To retrieve the details of Team who have not played any matches.

SQL> select \* from team where teamID not in(select teamid from match Union select playerId from match);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TeamID** | **BoardID** | **Name** | **Coach** | **Captain** |
| VCB01 | BID03 | RAINBOW | S.RAJESHKUMAR | MANIMARAN |
| MCB01 | BID05 | PANTHER | SARAVANAN | R.SUNILKUMAR |
| TUCB01 | BID06 | THUNDER | D ALEX | BARATHI |
| TICB01 | BID08 | KINGS | D ANAND | MATHAN |

* 1. To retrieve the teamid, boardid, teamname, and playername for a particular playerid given.

SQL> SELECT t.teamid, t.boardid, t.name, p.fname FROM team t JOIN player p ON t.teamid = p.teamid WHERE p.playerid = '66';

|  |  |  |  |
| --- | --- | --- | --- |
| **TeamID** | **BoardID** | **Name** | **FName** |
| VCB01 | BID03 | RAINBOW | Ganesh |

Result:

Thus the query using joins and writing subqueries has been done successfully.

**TASK 5**

**Writing Join Queries, equivalent, and/or recursive queries:**

(Tool: SQL/ Oracle, ALM: Flipped Classroom)

**Aim:** To Perform the advanced query processing and test its heuristics using designing of optimal correlated and nested sub queries such as finding summary statistics.

* 1. To retrieve all cricket boards and their teams.
  2. To list all matches along with the teams and their captains.
  3. To count the number of matches played by each team.
  4. To find all the players who are part of the team named " TIGER ROCK ".
  5. To retrieve all team details, including the count of winning matches for each team.
  6. To retrieve the total number of 'Tie' matches in a team-wise manner.
  7. To retrieve the team details who won the matches.
  8. To retrieve players and match details of players who are above 25 years old.
  9. To retrieve the details of Team who have not played any matches.
  10. To retrieve the teamid, boardid, teamname, and playername for a particular playerid given.
  11. To retrieve all cricket boards and their teams.

SQL> SELECT cb.Name AS CricketBoard, t.Name AS Team FROM CricketBoard cb JOIN Team t ON cb.BoardID = t.BoardID;

|  |  |
| --- | --- |
| **CricketBoard** | **Team** |
| Chennai Cricket Board | ABS EXPRESS |
| Chennai Cricket Board | AVG EXPRESS |
| Tiruvallur Cricket Board | ANGRY BARD |
| Tiruvallur Cricket Board | TIGER ROCK |
| Trichy Cricket Board | ROCK |
| Viluppuram Cricket Board | RAINBOW |
| Madurai Cricket Board | PANTHER |
| Tuticorin Cricket Board | THUNDER |
| Selam Cricket Board | EAGLE |
| Tiruppur Cricket Board | KINGS |

* 1. List all matches along with the teams and their captains.

SQL> SELECT m.match\_Date, m.Time1, m.matchID, t1.name AS team1\_name, t1.captain AS team1\_captain, t2.name AS team2\_name, t2.captain AS team2\_captain FROM match m JOIN team t1 ON m.teamID = t1.teamID JOIN team t2 ON m.playerID = t2.teamID;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Match\_Date** | **Time1** | **MatchID** | **team1\_name** | **team1\_captain** | **team2\_name** | **team2\_captain** |
| 22-JUN-2022 | 1.3 | M01 | ABS EXPRESS | SAMPATH KUMAR | ANGRY BARD | CINIL JOHN |
| 22-JUN-2022 | 8.3 | M02 | AVG EXPRESS | Y.JOHN | TIGER ROCK | BEN GEORGE |
| 24-JUN-2022 | 8.3 | M03 | ROCK | K.MUTHU | ANGRY BARD | CINIL JOHN |
| 25-JUN-2022 | 8.3 | M04 | ROCK | K.MUTHU | TIGER ROCK | BEN GEORGE |
| 04-APR-2023 | 7.3 | M05 | EAGLE | SRI HARI | AVG EXPRESS | Y.JOHN |

* 1. Count the number of matches played each team.

SQL> SELECT t.Name AS Team, COUNT(mt.TeamID) AS MatchesPlayed FROM Team t LEFT JOIN Match mt ON t.TeamID = mt.TeamID GROUP BY t.Name;

|  |  |
| --- | --- |
| **Team** | **MatchesPlayed** |
| ABS EXPRESS | 1 |
| ANGRY BARD | 0 |
| AVG EXPRESS | 1 |
| EAGLE | 1 |
| KINGS | 0 |
| PANTHER | 0 |
| RAINBOW | 0 |
| ROCK | 2 |
| THUNDER | 0 |
| TIGER ROCK | 0 |

* 1. To find all the players who are part of the team named "TIGER ROCK".

SQL> SELECT p.playerID, p.fname, p.teamID, t.coach, t.captain FROM player p JOIN team t ON p.teamID = t.teamID WHERE t.name = 'TIGER ROCK';

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PlayerID** | **FName** | **TeamID** | **Coach** | **Captain** |
| 102 | Akash | TCB02 | S.KANNAN | BEN GEORGE |
| 12 | Premkumar | TCB02 | S.KANNAN | BEN GEORGE |

* 1. To retrieve all team details, including the count of winning matches for each team.

SQL> SELECT t.TeamID, t.Name AS TeamName, t.Coach, t.Captain, COUNT(m.MatchID) AS WinningMatchCount FROM Team t LEFT JOIN Match m ON t.TeamID = substr(m.result,1,5) GROUP BY t.TeamID, t.Name, t.Coach, t.Captain;

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TeamID** | **TeamName** | **Coach** | **Captain** | **WinningMatchCount** |
| CCB01 | ABS EXPRESS | G.D.RAMESH | SAMPATH KUMAR | 1 |
| CCB02 | AVG EXPRESS | T.KARTHIK | Y.JOHN | 0 |
| MCB01 | PANTHER | SARAVANAN | R.SUNILKUMAR | 0 |
| SCB01 | EAGLE | SOMU | SRI HARI | 0 |
| TCB01 | ANGRY BARD | TOM BABU | CINIL JOHN | 2 |
| TCB02 | TIGER ROCK | S.KANNAN | BEN GEORGE | 0 |
| TICB01 | KINGS | D ANAND | MATHAN | 0 |
| TRICB01 | ROCK | K.PAUL | K.MUTHU | 0 |
| TUCB01 | THUNDER | D ALEX | BARATHI | 0 |
| VCB01 | RAINBOW | S.RAJESHKUMAR | MANIMARAN | 0 |

5.6 To retrieve the total number of 'Tie' matches in a team-wise manner.

SQL> SELECT t.Name AS TeamName, COUNT(\*) AS TotalTieMatches FROM Team t JOIN Match\_result mt ON t.TeamID = mt.TeamID JOIN Match\_result m ON mt.MatchID = m.MatchID WHERE m.Result = 'Tie' GROUP BY t.Name;

|  |  |
| --- | --- |
| **TeamName** | **TotalTieMatches** |
| ROCK | 1 |

5.7To retrieve the team details who won the matches.

SQL> select \* from team where teamID in (select mr.teamID from match\_result mr left join team t on mr.teamId=t.teamID where mr.result='Win');

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TeamID** | **BoardID** | **Name** | **Coach** | **Captain** |
| CCB01 | BID01 | ABS EXPRESS | G.D.RAMESH | SAMPATH KUMAR |
| TCB01 | BID02 | ANGRY BARD | TOM BABU | CINIL JOHN |
| TRICB01 | BID04 | ROCK | K.PAUL | K.MUTHU |

* 1. To retrieve players and match details of players who are above 25 years old.

SQL> SELECT p.PlayerID, p.FName AS PlayerName, p.Age, m.MatchID, m.match\_Date, m.Time1, m.Result FROM Player p, match m where p.playerID in(select playerID from player where age>25);

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PlayerID** | **PlayerName** | **Age** | **MatchID** | **Match\_Date** | **Time1** | **Result** |
| 1 | Raj | 27 | M01 | 22-JUN-2022 | 1.3 | TCB01 - WIN |
| 1 | Raj | 27 | M02 | 22-JUN-2022 | 8.3 | CCB01 - WIN |
| 1 | Raj | 27 | M03 | 24-JUN-2022 | 8.3 | TCB01 - WIN |
| 1 | Raj | 27 | M04 | 25-JUN-2022 | 8.3 | TRICB01 - WIN |
| 1 | Raj | 27 | M05 | 04-APR-2023 | 7.3 | Tie |

* 1. To retrieve the details of Team who have not played any matches.

SQL> select \* from team where teamID not in(select teamid from match Union select playerId from match);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TeamID** | **BoardID** | **Name** | **Coach** | **Captain** |
| VCB01 | BID03 | RAINBOW | S.RAJESHKUMAR | MANIMARAN |
| MCB01 | BID05 | PANTHER | SARAVANAN | R.SUNILKUMAR |
| TUCB01 | BID06 | THUNDER | D ALEX | BARATHI |
| TICB01 | BID08 | KINGS | D ANAND | MATHAN |

* 1. To retrieve the teamid, boardid, teamname, and playername for a particular playerid given.

SQL> SELECT t.teamid, t.boardid, t.name, p.fname FROM team t JOIN player p ON t.teamid = p.teamid WHERE p.playerid = '66';

|  |  |  |  |
| --- | --- | --- | --- |
| **TeamID** | **BoardID** | **Name** | **FName** |
| VCB01 | BID03 | RAINBOW | Ganesh |

**Result:**

Thus the query using Join Queries, equivalent, and/or recursive queries has been done successfully.

**TASK 6: Procedures, Function and Loops**

**Aim:** To write a programming using PL/SQL Procedures, Functions and loops on Number theory and business scenarios like.

1. Write a PL/SQL block that calculates the average age of players and displays the result.
2. Write a PL/SQL block that inserts a new player record into the Player table.
3. To create a function that returns the total number of teams in a particular Cricket Board.
4. To write a non-recursive PL/SQL procedure to retrieve even-numbered PlayerIDs registered for any tournament.

**Write a PL/SQL block that calculates the average age of players and displays the result.**

DECLARE

total\_age NUMBER := 0;

num\_players NUMBER := 0;

avg\_age NUMBER := 0;

BEGIN

-- Using a cursor to loop through all players

FOR player\_rec IN (SELECT Age FROM Player) LOOP

total\_age := total\_age + player\_rec.Age; -- Summing up the ages

num\_players := num\_players + 1; -- Counting the number of players

END LOOP;

-- Calculating the average age

IF num\_players > 0 THEN

avg\_age := total\_age / num\_players;

END IF;

-- Displaying the result

DBMS\_OUTPUT.PUT\_LINE('Total Players: ' || num\_players);

DBMS\_OUTPUT.PUT\_LINE('Total Age: ' || total\_age);

DBMS\_OUTPUT.PUT\_LINE('Average Age: ' || avg\_age);

END;

Output:

Total Players: 14

Total Age: 342

Average Age: 24.42

**Write a PL/SQL block that inserts a new player record into the Player table.**

DECLARE

v\_PlayerID VARCHAR(6) := &PlayerID’; -- You can generate a unique PlayerID as needed

v\_TeamID VARCHAR(6) := '&TEAMID'; -- Replace with the actual TeamID

v\_FName VARCHAR(30) := '&Fname';

v\_LName VARCHAR(30) := '&Lname';

v\_Age NUMBER(5,2) := &age;

v\_DateofBirth DATE := TO\_DATE('&DOB', 'YYYY-MM-DD'); -- Replace with the actual DateofBirth

v\_PlayingRole VARCHAR(25) := '&PlayingRole';

v\_email VARCHAR(40) := '&email';

v\_contact\_no NUMBER := &phone; -- Replace with the actual contact number

BEGIN

INSERT INTO Player (PlayerID, TeamID, FName, LName, Age, DateofBirth, PlayingRole, email, contact\_no)

VALUES (v\_PlayerID, v\_TeamID, v\_FName, v\_LName, v\_Age, v\_DateofBirth, v\_PlayingRole, v\_email, v\_contact\_no);

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Player record inserted successfully.');

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

ROLLBACK;

END;

/

Enter the PlayerID: 676

Enter the TeamID: CCB01

Enter the FName: Rahul

Enter the LName: Sharma

Enter the Age: 23

Enter the DateofBirth: 17-07-1999

Enter the PlayingRole: AllRounder

Enter the email: rahulsharma@gmail.com

Enter the Contact\_no: 9797181815

Player record inserted successfully.

**To create a function that returns the total number of teams in a particular Cricket Board.**

CREATE OR REPLACE FUNCTION GetTotalTeamsInBoard(BoardID VARCHAR2) RETURN NUMBER IS

v\_TotalTeams NUMBER := 0;

BEGIN

SELECT COUNT(\*) INTO v\_TotalTeams FROM Team WHERE BoardID = BoardID;

RETURN v\_TotalTeams;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

-- Handle the case when the board doesn't exist or has no teams

RETURN 0;

WHEN OTHERS THEN

-- Handle other exceptions as needed

RETURN -1; -- Return a negative value to indicate an error

END GetTotalTeamsInBoard;

/

**Function successfully created.**

**SQL>**

Declare

**number res;**

**Begin  
res:=** GetTotalTeamsInBoard(‘BID01’);

DBMS\_OUTPUT.PUT\_LINE(‘No of teams: ‘||res);

END;

/

No of teams: 2

**To write a non-recursive PL/SQL procedure to retrieve even-numbered PlayerIDs registered for any tournament.**

CREATE OR REPLACE PROCEDURE GetEvenNumberedPlayerIDs IS

BEGIN

FOR player\_rec IN ( SELECT PlayerID FROM Player WHERE TO\_NUMBER(PlayerID) MOD 2 = 0)

LOOP

DBMS\_OUTPUT.PUT\_LINE('Even-Numbered PlayerID: ' || player\_rec.PlayerID);

END LOOP;

END GetEvenNumberedPlayerIDs;

/

**Result:**

Thus the PL/SQL Procedures, Functions and loops on Number theory and business scenarios experiment was successfully completed and results are verified.

**TASK 7**: **Triggers, Views and Exceptions**

**Aim:**

To Conduct events, views, exceptions on CRUD operations for restricting phenomenon.

a) To create a trigger in PL/SQL that automatically inserts a new record in the match\_result table when a new record is inserted into the match table.

b) To create a view that displays the details of players along with their team details.

c) To write a non-recursive PL/SQL procedure to retrieve even-numbered PlayerIDs registered for any tournament.

**To create a trigger in PL/SQL that automatically inserts a new record in the match\_result table when a new record is inserted into the match table.**

CREATE OR REPLACE TRIGGER insert\_match\_result

AFTER INSERT ON match

FOR EACH ROW

BEGIN

INSERT INTO match\_result (MatchID, TeamID, Result)

VALUES (:new.MatchID, :new.TeamID, 'Pending'); -- Assuming a default value of 'Pending' for Result

END;

/

**To create a view that displays the details of players along with their team details.**

SQL> CREATE VIEW PlayerTeamDetails AS SELECT p.playerID, p.fname AS PlayerName, p.teamID, p.coach AS PlayerCoach, p.captain AS PlayerCaptain, (SELECT t.name FROM team t WHERE t.teamID = p.teamID) AS TeamName, (SELECT t.coach FROM team t WHERE t.teamID = p.teamID) AS TeamCoach, (SELECT t.captain FROM team t WHERE t.teamID = p.teamID) AS TeamCaptain FROM player p;

SQL> Select \* from PlayerTeamDetails;

**To write a non-recursive PL/SQL procedure to retrieve even-numbered PlayerIDs registered for any tournament.**

CREATE OR REPLACE PROCEDURE GetEvenPlayerIDsForTournament(in\_tournament\_id NUMBER, out\_even\_player\_ids SYS.ODCINUMBERLIST) AS

BEGIN

out\_even\_player\_ids := SYS.ODCINUMBERLIST(); -- Initialize the collection

-- Populate the collection with even-numbered PlayerIDs for the specified tournament

FOR player\_rec IN (SELECT PlayerID FROM Player WHERE TournamentID = in\_tournament\_id AND MOD(PlayerID, 2) = 0) LOOP

out\_even\_player\_ids.EXTEND;

out\_even\_player\_ids(out\_even\_player\_ids.COUNT) := player\_rec.PlayerID;

END LOOP;

END;

/

DECLARE

tournament\_id NUMBER := 123; -- Replace with the desired tournament ID

even\_player\_ids SYS.ODCINUMBERLIST;

BEGIN

GetEvenPlayerIDsForTournament(tournament\_id, even\_player\_ids);

-- You can now use the even\_player\_ids collection as needed.

-- For example, to print the even PlayerIDs:

FOR i IN 1..even\_player\_ids.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Even PlayerID: ' || even\_player\_ids(i));

END LOOP;

END;

/

**Result:**

Thus the Triggers, Views and Exceptions experiment was successfully completed results are verified.

**TASK 8**

**CRUD operations in Document databases**

**AIM:**

To Perform Mongoose using NPM design on MongoDB designing document database and performing CRUD operations like creating, inserting, querying, finding, removing operations

**STEPS:**

Step 1: Install Mongo db using following link

https://www.mongodb.com/try/download/community

Step 2: install Mongosh using the below link

https://www.mongodb.com/docs/mongodb-shell/#download-and-install-mongosh

Step 3: To add the MongoDB Shell binary's location to your PATH environment variable:

3.1 Open the Control Panel.

3.2 In the System and Security category, click System.

3,3 Click Advanced system settings. The System Properties modal displays.

3.4 Click Environment Variables.

3.5 In the System variables section, select path and click Edit. The Edit environment

variable modal displays.

3.6 Click New and add the filepath to your mongosh binary.

3.7 Click OK to confirm your changes. On each other modal, click OK to confirm

your changes.

Step 4: To confirm that your PATH environment variable is correctly configured to find

mongosh, open a command prompt and enter the mongosh --help command. If your

PATH is configured correctly, a list of valid commands displays.

Step 5: Open mongo shell 4.0 from c:\programfiles\mongoDB\server\bin\mongod.exe

Step 6: Type the CRUD(CREATE READ UPDATE DELETE) COMMANDS GIVEN IN

TEXT FILE.

**CRUD OPERATIONS:**

db.createCollection("CricketBorad")

{ "ok" : 1 }

> db.CricketBoard.insertOne({BoardID: "BID01", Name:"Chennai Cricket Board", Address:"Chennai",Phone:9988776699});

{

"acknowledged" : true,

"insertedId" : ObjectId("651cf1726ebbfe7993adf909")

}

db.mylab.find({BoardID:"BID01"})

{ "\_id" : ObjectId("651cf1726ebbfe7993adf909"), "BoardID" : "BID01", "Name" : "Chennai Cricket Board", "Address" : "Chennai", "Phone" : 9988776699 }

db.CricketBoard.insertMany([{BoardID:"BID02",Name:"Tiruvallur Cricket Board",Address:"Chennai",Phone:9977886699},{BoardID:"BID03",Name:"Viluppuram Cricket Board",Address:"Viluppuram",Phone:9966886699},{BoardID:"BID04",Name:"Trichy Cricket Board",Address:"Trichy",Phone:9955886699},{BoardID:"BID05",Name:"Madurai Cricket Board",Address:"Madurai",Phone:9944886699}]);

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("651ceee36ebbfe7993adf904"),

ObjectId("651ceee36ebbfe7993adf905"),

ObjectId("651ceee36ebbfe7993adf906"),

ObjectId("651ceee36ebbfe7993adf907")

]

}

db.CricketBoard.find()

{ "\_id" : ObjectId("651ceee36ebbfe7993adf904"), "BoardID" : "BID02", "Name" : "Tiruvallur Cricket Board", "Address" : "Chennai", "Phone" : 9977886699 }

{ "\_id" : ObjectId("651ceee36ebbfe7993adf905"), "BoardID" : "BID03", "Name" : "Viluppuram Cricket Board", "Address" : "Viluppuram", "Phone" : 9966886699 }

{ "\_id" : ObjectId("651ceee36ebbfe7993adf906"), "BoardID" : "BID04", "Name" : "Trichy Cricket Board", "Address" : "Trichy", "Phone" : 9955886699 }

{ "\_id" : ObjectId("651ceee36ebbfe7993adf907"), "BoardID" : "BID05", "Name" : "Madurai Cricket Board", "Address" : "Madurai", "Phone" : 9944886699 }

{ "\_id" : ObjectId("651cf1726ebbfe7993adf909"), "BoardID" : "BID01", "Name" : "Chennai Cricket Board", "Address" : "Chennai", "Phone" : 9988776699 }

db.CricketBoard.find().pretty()

{

"\_id" : ObjectId("651ceee36ebbfe7993adf904"),

"BoardID" : "BID02",

"Name" : "Tiruvallur Cricket Board",

"Address" : "Chennai",

"Phone" : 9977886699

}

{

"\_id" : ObjectId("651ceee36ebbfe7993adf905"),

"BoardID" : "BID03",

"Name" : "Viluppuram Cricket Board",

"Address" : "Viluppuram",

"Phone" : 9966886699

}

{

"\_id" : ObjectId("651ceee36ebbfe7993adf906"),

"BoardID" : "BID04",

"Name" : "Trichy Cricket Board",

"Address" : "Trichy",

"Phone" : 9955886699

}

{

"\_id" : ObjectId("651ceee36ebbfe7993adf907"),

"BoardID" : "BID05",

"Name" : "Madurai Cricket Board",

"Address" : "Madurai",

"Phone" : 9944886699

}

{

"\_id" : ObjectId("651cf1726ebbfe7993adf909"),

"BoardID" : "BID01",

"Name" : "Chennai Cricket Board",

"Address" : "Chennai",

"Phone" : 9988776699

}

db.CricketBoard.deleteOne({BoardID:"BID03"})

{ "acknowledged" : true, "deletedCount" : 1 }

db.CricketBoard.find().pretty()

{

"\_id" : ObjectId("651ceee36ebbfe7993adf904"),

"BoardID" : "BID02",

"Name" : "Tiruvallur Cricket Board",

"Address" : "Chennai",

"Phone" : 9977886699

}

{

"\_id" : ObjectId("651ceee36ebbfe7993adf906"),

"BoardID" : "BID04",

"Name" : "Trichy Cricket Board",

"Address" : "Trichy",

"Phone" : 9955886699

}

{

"\_id" : ObjectId("651ceee36ebbfe7993adf907"),

"BoardID" : "BID05",

"Name" : "Madurai Cricket Board",

"Address" : "Madurai",

"Phone" : 9944886699

}

{

"\_id" : ObjectId("651cf1726ebbfe7993adf909"),

"BoardID" : "BID01",

"Name" : "Chennai Cricket Board",

"Address" : "Chennai",

"Phone" : 9988776699

}

**Result:**

Thus CRUD using NPM design on MongoDB designing document database and performing CRUD operations like creating, inserting, querying, finding, removing operations are performed.

**TASK 9**

**CRUD operations in Graph databases**

**AIM:**

To perform CRUD operations like creating, inserting, querying, finding, deleting operations on graph spaces.

**The steps to get started with Neo4j's Aura Graph Database:**

**Step1:** Copy and paste the following link into your web browser: https://neo4j.com/cloud/platform/aura-graph-database/?ref=docs-get-started-dropdown

**Step2:** Click on "Start Free."

**Step3:** Choose the option to "Continue with Google."

**Step4:** Click the "Open" button.

**Step5:** After clicking "Open," a text file will be automatically downloaded. This file contains your user ID and password details.

**Step6:** Copy the password from the downloaded text file and paste it where required.

**Step7:** Close the "Get started with Neo4j with beginner guides" if it's open.

**Sep8:** You're now ready to begin practicing with the Graph Database.

## Create Node with Properties

Properties are the key-value pairs using which a node stores data. Create a node with properties using the CREATE clause and need to specify these properties separated by commas within the flower braces “{ }”.

**Syntax**

CREATE (node:label { key1: value, key2: value, . . . . . . . . . }) return node

To verify the creation of the node, type and execute the following query in the dollar prompt.

**Syntax:**

MATCH (n) RETURN n

**Creating Relationships**

To create a relationship using the CREATE clause and specify relationship within the square braces “[ ]” depending on the direction of the relationship it is placed between hyphen “ - ” and arrow “ → ” as shown in the following syntax.

**Syntax:**

CREATE (node1)-[:RelationshipType]->(node2)

**Syntax:**

MATCH (a:LabeofNode1), (b:LabeofNode2)

WHERE a.name = "nameofnode1" AND b.name = " nameofnode2"

CREATE (a)-[: Relation]->(b) RETURN a,b

**Deleting a Particular Node**

To delete a particular node and need to specify the details of the node in the place of “n” in the above query.

**Syntax:**

MATCH (node:label {properties . . . . . . . . . . }) DELETE node

Create a graph database for student course registration, create student and dept node and insert values of properties.

**Create a CrickerBoard Node:**

create(cb:CricketBoard{BoardID:'BID01',Name:'Chennai Cricket Board', Address:'Chennai', Phone:9988776699}) return cb

**Create Team Nodes:**

create(t1:Team{teamID:'CCB01',BoardID:'BID01',name:'ABS EXPRESS', Coach:'G.D.RAMESH', Captain:'SAMPATH KUMAR'}) return t1

create(t2:Team{teamID:'CCB02',BoardID:'BID01',name:'AVG EXPRESS',Coach: 'T.KARTHIKH', Captain:'Y.JOHN'}) return t2

**Create Player Nodes:**

create(p1:Player{PlayerID:'1',TeamID:'CCB01',Name:'Raj',Age:23,DateofBirth:'29-JUN-1996', PlayingRole:'Bowler',email:'rajn@gmail.com'}) return p1

create(p2:Player{PlayerID:'33',TeamID:'CCB01',Name:'Anand',Age:23,DateofBirth:'02-JAN-1999', PlayingRole:'Batsman',email:'balajid@gmail.comm'}) return p2

create(p3:Player{PlayerID:'65',TeamID:'CCB02',Name:'Suresh',Age:27,DateofBirth:'02-JUN-1996', PlayingRole:'Batsman',email:'sureshd@gmail.comm'}) return p3

create(p4:Player{PlayerID:'75',TeamID:'CCB02',Name:'Rohit',Age:33,DateofBirth:'02-JUN-1991', PlayingRole:'Batsman',email:'srohit@gmail.comm'}) return p4

**Creating Relationship among CricketBoard and Teams:**

match(cb:CricketBoard{BoardID:'BID01'}),(t1:Team{teamID:'CCB01'}) create(cb)-[r:has]->(t1) return cb,r,t1

match(cb:CricketBoard{BoardID:'BID01'}),(t2:Team{teamID:'CCB02'}) create(cb)-[r:has]->(t2) return cb,r,t2

**Creating Relationship among Players and Teams:**

match(p1:Player{PlayerID:'1'}),(t1:Team{teamID:'CCB01'}) create(p1)-[r1:playfor]->(t1) return p1,r1,t1

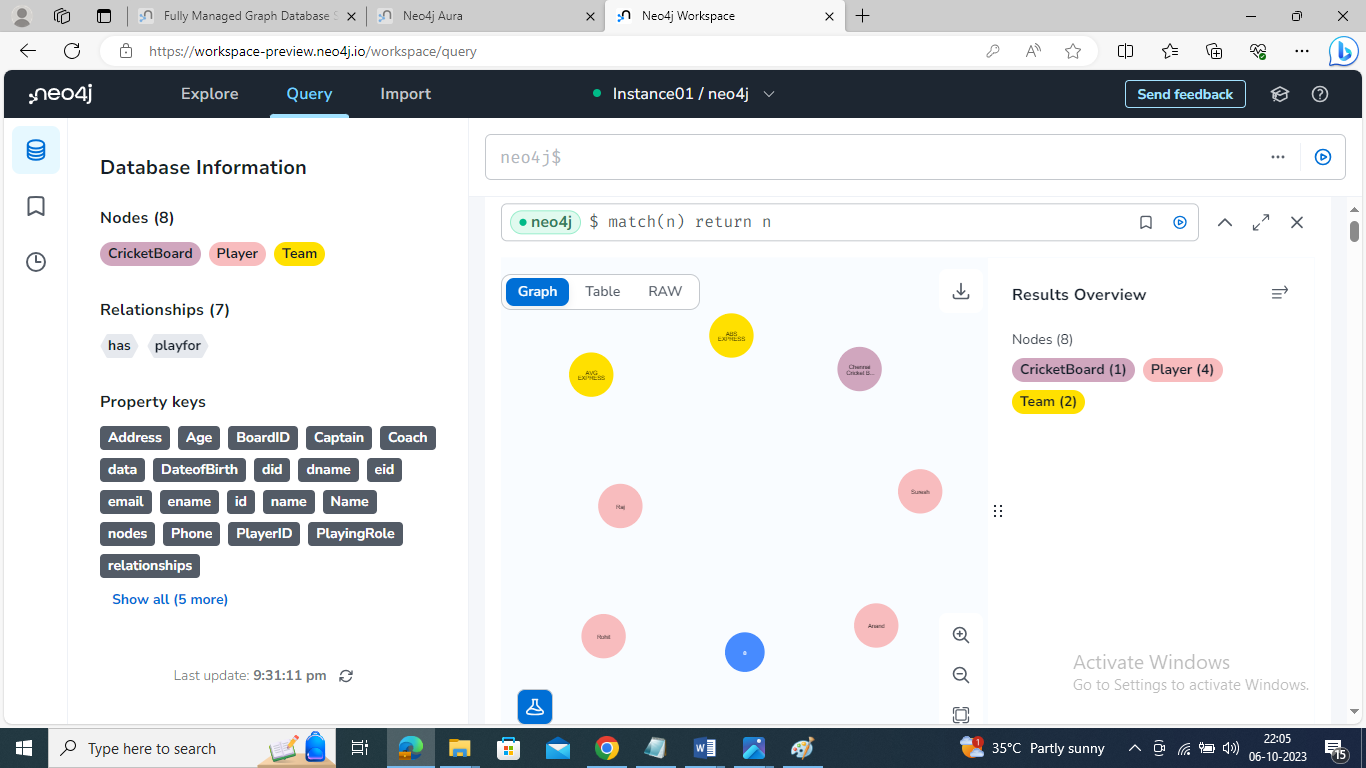
match(p2:Player{PlayerID:'33'}),(t1:Team{teamID:'CCB01'}) create(p2)-[r2:playfor]->(t1) return p2,r2,t1

match(p3:Player{PlayerID:'65'}),(t2:Team{teamID:'CCB02'}) create(p3)-[r3:playfor]->(t2) return p3,r3,t2

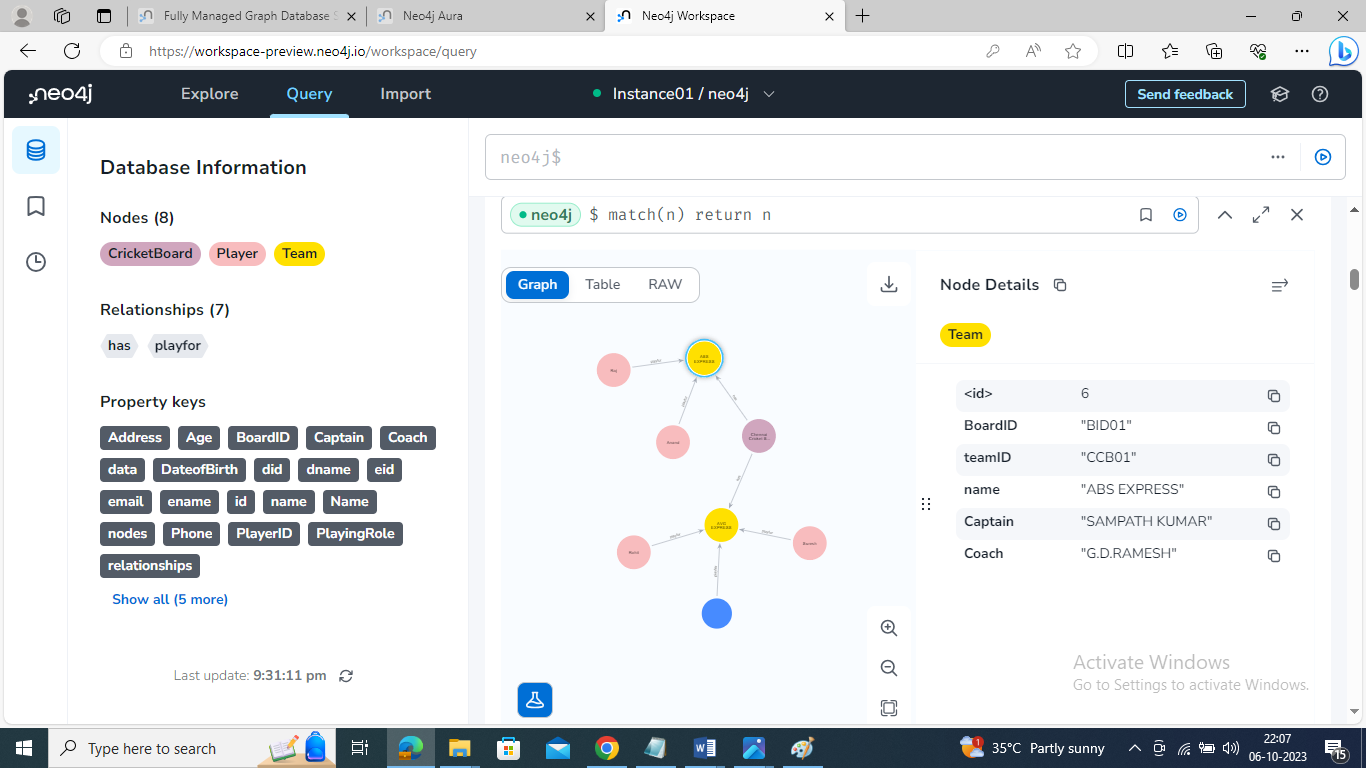
match(p4:Player{PlayerID:'75'}),(t2:Team{teamID:'CCB02'}) create(p3)-[r4:playfor]->(t2) return p4,r4,t2

**Display All nodes:** match(n) return n

**Output:**

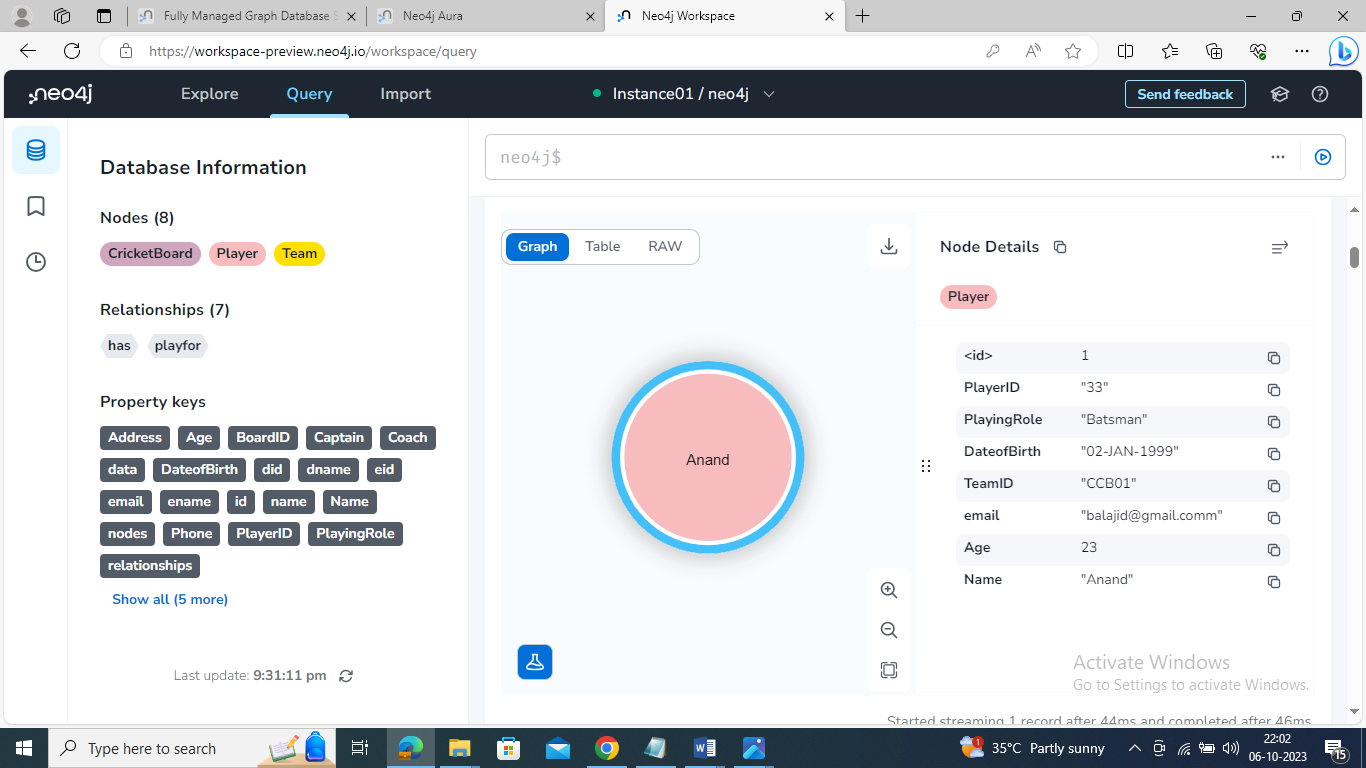
****

**OUTPUT:**



**Retrieve particular player details:**

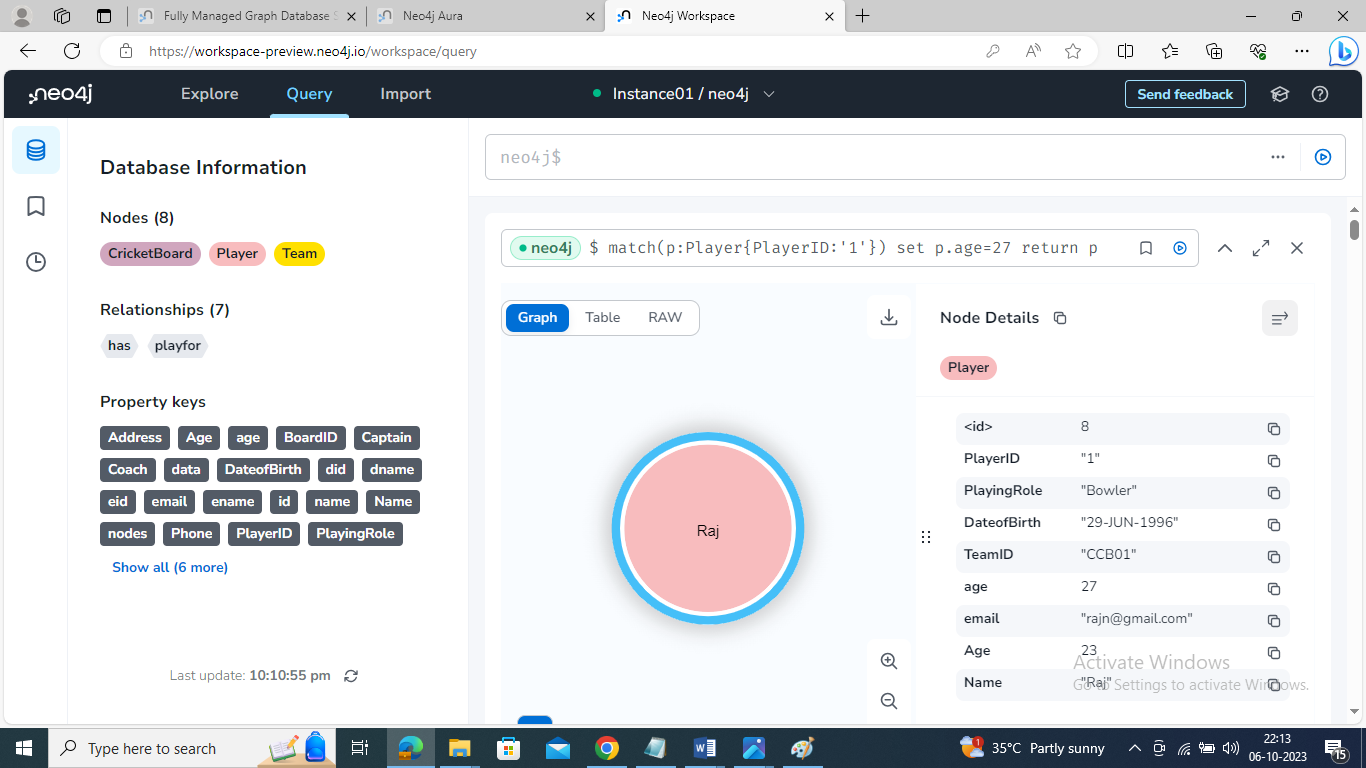
match(p:Player{PlayerID:'33'}) return p



**Update particular player details:**

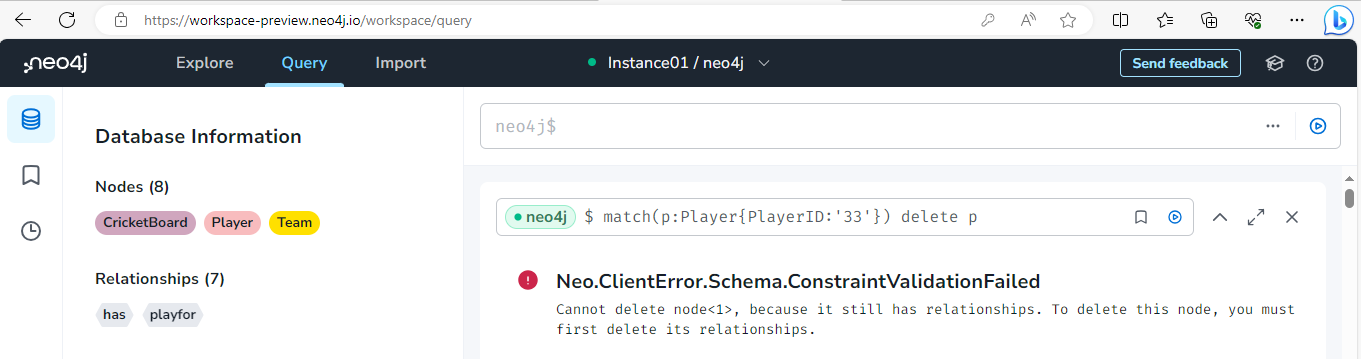
match(p:Player{PlayerID:'1'}) set p.age=27 return p

**Output:**

****

**Delete particular player from the team:**

match(p:Player{PlayerID:'33'}) delete p

****

**Result:**

Thus the CRUD operations like creating, inserting, querying, finding, deleting operations on graph spaces were executed successfully.

**Task 10**

**Normalizing databases using functional dependencies upto Third Normal Form**

**Aim:** To normalize the below relation and create the simplified table with suitable constraint.

CricketBoard(BoardID, Name, Address, Contact\_No, TeamID, TName, Coach, Captain, PlayerID, PFName, PLName, Age, PDateofBirth, PlayingRole, email, contact\_no, Batting, Bowling, MatchID, Match\_Date, Time1, Result, GroundID, GName, Location, Capacity, UmpireID, UFName, ULName, UAge, UDateofBirth, Country, Uemail, Ucontact\_no).

1. Apply the functional dependency, normalize to 1NF
2. Normalize the relations using FD+ and α+.
3. Find the minimal cover, canonical cover.
4. Normalize to 2NF, add/alter constraints if necessary.
5. Normalize to 3NF, add/alter constraints if necessary.

**Procedure:**

Normalize the given relation and create simplified tables with suitable constraints, we need to identify the functional dependencies and separate them into different tables. Normalization involves breaking down the data into smaller, related tables to minimize data redundancy and maintain data integrity. Let's identify the functional dependencies:

**Functional Dependency:**

BoardID → Name, Address, Contact\_No

TeamID → TName, Coach, Captain

PlayerID → PFName, PLName, Age, PDateofBirth, PlayingRole, email, contact\_no, Batting, Bowling

MatchID → Match\_Date, Time1, Result, GroundID

GroundID → GName, Location, Capacity

UmpireID → UFName, ULName, UAge, UDateofBirth, Country, Uemail, Ucontact\_no

Now, we can create simplified tables:

CricketBoard (BoardID [PK], Name, Address, Contact\_No)

CricketTeam (TeamID [PK], TName, Coach, Captain)

CricketPlayer (PlayerID [PK], TeamID [FK], PFName, PLName, Age, PDateofBirth, PlayingRole, email, contact\_no, Batting, Bowling)

CricketMatch (MatchID [PK], TeamID [FK], Match\_Date, Time1, Result, GroundID [FK])

CricketGround (GroundID [PK], GName, Location, Capacity)

CricketUmpire (UmpireID [PK], UFName, ULName, UAge, UDateofBirth, Country, Uemail, Ucontact\_no)

In these tables, [PK] denotes the primary key, [FK] denotes the foreign key, and suitable constraints should be added to maintain data integrity.

**Create tables for all non-prime attributes using α+**

α+ (Alpha Plus) allows to group attributes based on their functional dependencies and candidate keys. And create tables for each set of attributes that functionally depend on a candidate key. The candidate keys in this case are BoardID, TeamID, PlayerID, MatchID, and UmpireID.

Board Table: BoardID (PK), Name, Address, Contact\_No

Team Table: TeamID (PK), TName, Coach, Captain

Player Table: PlayerID (PK), TeamID (FK), PFName, PLName, Age, PDateofBirth, PlayingRole,

Email, contact\_no, Batting, Bowling

Match Table: MatchID (PK), TeamID (FK), Match\_Date, Time1, Result

Ground Table: GroundID (PK), GName, Location, Capacity

Umpire Table: UmpireID (PK), UFName, ULName, UAge, UDateofBirth, Country, Uemail, Ucontact\_no

Create additional tables to represent transitive dependencies.

Already addressed transitive dependencies in previous normalization steps by introducing the MatchVenue table for the transitive dependency between MatchID and GroundID through the Result attribute.

MatchVenue Table: MatchID (PK, FK), GroundID (FK)

**First Normal Form:**

The given relation into the First Normal Form (1NF), to need to ensure that each attribute (column) contains atomic (indivisible) values, and there are no repeating groups or arrays.

Based on the provided relation, it appears that each attribute already contains atomic values, so there are no repeating groups to eliminate.

**Second Normal Form:**

To determine whether the given relation is in the Second Normal Form (2NF), we need to check two conditions:

The relation must already be in 1NF (First Normal Form).

All non-prime attributes (attributes not part of any candidate key) must be fully functionally dependent on the entire primary key.

First, let's identify the potential candidate key(s) from the given relation based on functional dependencies:

It appears that the potential candidate keys could be:

1. BoardID
2. TeamID
3. PlayerID
4. MatchID
5. UmpireID

Next, we need to check if all non-prime attributes are fully functionally dependent on their respective candidate key(s).

**Third Normal Form:**

To determine whether the given relation is in the Third Normal Form (3NF), need to check two conditions:

1. The relation must already be in the Second Normal Form (2NF).
2. There should be no transitive dependencies between non-prime attributes and candidate keys.

The given relation satisfies the conditions of the Second Normal Form (2NF). Now, let's check for transitive dependencies:

Now, let's analyze each functional dependency and check for transitive dependencies:

BoardID → Name, Address, Contact\_No

There are no transitive dependencies in this case, as Name, Address, and Contact\_No are directly dependent on BoardID.

TeamID → TName, Coach, Captain

There are no transitive dependencies here either, as TName, Coach, and Captain are directly dependent on TeamID.

PlayerID → PFName, PLName, Age, PDateofBirth, PlayingRole, email, contact\_no, Batting, Bowling

There are no transitive dependencies for PlayerID, as all the mentioned attributes are directly dependent on PlayerID.

MatchID → Match\_Date, Time1, Result, GroundID

There is a transitive dependency between MatchID and GroundID through the Result attribute. To resolve this, we create a new table called MatchVenue:

MatchVenue (MatchID [PK], GroundID [FK])

GroundID → GName, Location, Capacity

There are no transitive dependencies for GroundID, as GName, Location, and Capacity are directly dependent on GroundID.

UmpireID → UFName, ULName, UAge, UDateofBirth, Country, Uemail, Ucontact\_no

There are no transitive dependencies for UmpireID, as UFName, ULName, UAge, UDateofBirth, Country, Uemail, and Ucontact\_no are directly dependent on UmpireID.

With the introduction of the MatchVenue table to resolve the transitive dependency, the relation now satisfies the conditions of the Third Normal Form (3NF).

**Result:**

Thus the normalization of the given relation is created the simplified tables with suitable constraint successfully.

**TASK 11**

**Menus, Forms and Reports**

**Aim:**

To designing an application with Oracle Forms, Menus and Report Builder involves creating a user interface (UI) using Forms and generating reports using Report Builder.

**Install Oracle Forms and Report Builder:**

Ensure that the Oracle Forms and Report Builder installed on your development machine.

**Design the Data Model:**

In Oracle Forms to define the data model that connects to the database schema. Use the Data Block Wizard to create data blocks that represent the tables or views. Ensure that the set up data blocks for all the data to need to work with in the application.

**Create Menus**

Menus provide the navigation structure for application. To create menus in Oracle Forms:

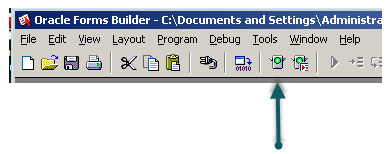
**Step1:** Open Oracle Forms Builder.

**Step2:** Create a new form for menu or use an existing one.

**Step3:** Add menu items for each function or feature of application.

**Step4:** Define the menu hierarchy and assign triggers or procedures to handle menu item actions.

**Step5:** Compile and run the menu form to test the navigation.



**Design Forms**

Forms are used to capture, display, and edit data. To design forms in Oracle Forms:

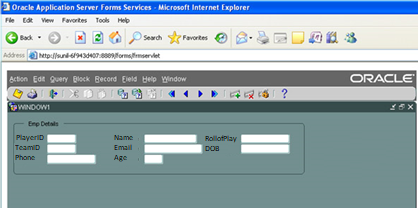
**Step1:** Create a new form for each major component of application.

**Step2:** Add form elements like text fields, buttons, and lists to forms.

**Step3:** Use the Property Palette to configure the properties of form elements and data blocks.

**Step4:** Write PL/SQL code to handle business logic and data validation.

**Step5:** Test the forms within the Forms Builder environment.



**Create Reports**

Reports provide a way to present data from application. To create reports using Oracle Report Builder:

**Step1:** Open Oracle Report Builder.

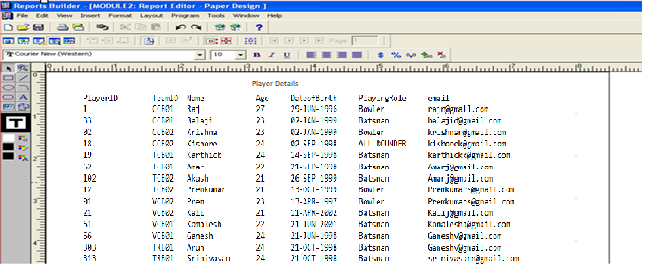
**Step2:** Create a new report or use an existing one.

**Step3:** Define the data source for the report (e.g., a database query or PL/SQL procedure).

**Step4:** Design the report layout, including headers, footers, and data columns.

**Step5:** Add report parameters if needed to allow users to customize the report.

**Step6:** enerate and preview the report to ensure it meets the requirements.



**RESULT**

Thus designing an application with Oracle Forms, Menus and Report Builder involves creating a user interface (UI) using Forms and generating reports using Report Builder has done successfully.