A database-management system (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the database, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

View of Data

A database system is a collection of interrelated files and a set of programs that allow users to access and modify these files. A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.

Data Abstraction

* Physical level. The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.
* Logical level. The next-higher level of abstraction describes what data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures.
* View level. The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

**Instances and Schemas**

Databases change over time as information is inserted and deleted. The collection of **information** stored in the database **at a particular moment** is called an instance of the database.

The **overall design** of the database is called the **database schema**. Schemas are changed infrequently.

Database systems have several schemas, partitioned according to the levels of abstraction.

The physical schema describes the database design at the physical level, while the logical schema describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called subschemas that describe different views of the database.Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema.

**Data Models**

Underlying the structure of a database is the data model:

A collection of conceptual tools for

* Describing data,
* Data relationships,
* Data semantics, and
* Consistency constraints.

The entity-relationship model and the relational model, both provide a way to describe the design of a database at the logical level.

The Entity-Relationship(E-R) Model

The entity-relationship data model is based on a perception of a real world that consists of a collection of basic objects, called entities, and of relationships among these objects. An entity is a “thing” or “object” in the real world that is distinguishable from other objects.

**Entities** are described in a database by a set of **attributes**. For example, the attributes account-number and balance may describe one particular account in a bank, and they form attributes of the account entity set.

A **relationship** is an association among several entities. For example, a depositor relationship associates a customer with each account that she has. The set of all entities of the same type and the set of all relationships of the same type are termed an entity set and relationship set, respectively.

The overall logical structure (schema) of a database can be expressed graphically by an E-R diagram, which is built up from the following components:

* Rectangles, which represent entity sets
* Ellipses, which represent attributes
* Diamonds, which represent relationships among entity sets
* Lines, which link attributes to entity sets and entity sets to relationships



**Relational Model**

The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name.

The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

The relational model is at a lower level of abstraction than the E-R model. Database designs are often carried out in the E-R model, and then translated to the relational model;

The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type.

**Object-oriented data model**

The object-oriented data model is another data model that has seen increasing attention. The object-oriented model can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model.

Semi-structured data models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The extensible markup language (XML) is widely used to represent semi-structured data.

**Database Languages**

A database system provides a **DDL** to specify the database schema and a **DML** to express database queries and updates. In practice, the DDL and DML’s form parts of a single database language, such as SQL.

**Data-Definition Language:**

create table account(

account-number char(10),

balance integer

)

Execution of the above DDL statement creates the account table. In addition, it updates a special set of tables called the **data dictionary** or **data directory**.

A data dictionary contains metadata—that is, data about data. The schema of a table is an example of metadata. A database system consults the data dictionary before reading or modifying actual data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain consistency constraints.

For example, suppose the balance on an account should not fall below $100. The DDL provides facilities to specify such constraints. The database systems check these constraints every time the database is updated.

**Data manipulation is**

* The **retrieval** of information stored in the database
* The **insertion** of new information into the database
* The **deletion** of information from the database
* The **modification** of information stored in the database

A data-manipulation language (DML) is a language that enables users to access or manipulate data as organized by the appropriate data model.

There are basically two types:

* Procedural DMLs require a user to specify what data are needed and how to get those data.
* Declarative DMLs (also referred to as nonprocedural DMLs) require a user to specify what data are needed without specifying how to get those data.

The portion of a DML that involves information retrieval is called a query language.

The levels of abstraction apply not only to defining or structuring data, but also to manipulating data.

The query processor component of the database system translates DML queries into sequences of actions at the physical level of the database system.

Database Access from Application Programs

Application programs are programs that are used to interact with the database. Application programs are usually written in a host language, such as Java. To access the database, DML statements need to be executed from the host language.

There are two ways to do this:

* By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database, and retrieve the results.
  + The Java Database Connectivity (JDBC) standard provides such features to the Java language.
* By extending the host language syntax to embed DML calls within the host language program.
  + Usually, a special character prefaces DML calls, and a preprocessor, called the DML precompiler, converts the DML statements to normal procedure calls in the host language.

Transaction Management

Consider an example of a funds transfer, in which one account (say A) is debited and another account (say B) is credited. Clearly, it is essential that either both the credit and debit occur, or that neither occur.

* This all-or-none requirement is called **atomicity**.
* It is essential that the execution of the funds transfer preserve the consistency of the database. That is, the value of the sum A + B must be preserved. This correctness requirement is called **consistency**.
* After the successful execution of a funds transfer, the new values of accounts A and B must persist, despite the possibility of system failure. This persistence requirement is called **durability**.

It is the **programmer’s responsibility** to define properly the various transactions, so that each preserves the **consistency** of the database.

Ensuring the **atomicity and durability** properties is the responsibility of the **database system itself**—specifically, of the transaction-management component.

Because of various types of failure, a transaction may not always complete its execution successfully. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database. Thus, the database must be restored to the state in which it was before the transaction in question started executing.

The database system must therefore perform failure recovery, that is, detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

Finally, when several transactions update the database concurrently, the consistency of data may no longer be preserved, even though each individual transaction is correct. It is the responsibility of the concurrency-control manager to control the interaction among the concurrent transactions, to ensure the consistency of the database.

A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates.

However, during the execution of a transaction, the temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.

**Database System Structure**

The functional components of a database system can be broadly divided into the **storage manager** and the **query processor components**.

The storage manager is important because databases typically require a large amount of storage space.

Since the main memory of computers cannot store this much information, the information is stored on disks. Data are moved between disk storage and main memory as needed. Since the movement of data to and from disk is slow relative to the speed of the central processing unit, it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory.

The query processor is important because it helps the database system simplify and facilitate access to data. High-level views help to achieve this goal; with them, users of the system are not be burdened unnecessarily with the physical details of the implementation of the system. However, quick processing of updates and queries is important. It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

Storage Manager

The storage manager is responsible for storing, retrieving, and updating data in the database.

The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands.

A storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager.

The storage manager components include:

* **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
* **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
* **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
* **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

The Query Processor

The query processor components include

* **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.
* **DML compiler**, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.
* **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.

Database Users and User Interfaces

There are four different types of database-system users

* **Naive users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously.
  + For example, to see her account balance over internet user may access a form, where she enters her account number.
  + The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form.
* **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.
* **Sophisticated users** interact with the system without writing programs. Instead, they form their requests in a database query language using query processor.
* **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems; that store data with complex data types (for example, graphics data and audio data)

Database Administrator

Person who has central control of both the data and the programs that access those data. The functions of a DBA include:

* Schema definition. The DBA creates the original database schema by executing a set of data definition statements in the DDL.
* Storage structure and access-method definition.
* Schema and physical-organization modification.
  + to reflect the changing needs of the organization, or
  + To alter the physical organization to improve performance.
* Granting of authorization for data access.
  + The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
* Routine maintenance. Examples of the database administrator’s routine maintenance activities are:
  + Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
  + Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
  + Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

**Keys**

**[http://www.w3schools.com/sql/sql\_unique.asp]**

[<http://rdbms.opengrass.net/2_Database%20Design/2.1_TermsOfReference/2.1.2_Keys.html>]

Super Key

A Super key is any combination of fields within a table that uniquely identifies each record within that table.

Candidate Key

A candidate is a subset of a super key. A candidate key is a single field or the least combination of fields that uniquely identifies each record in the table. The least combination of fields distinguishes a candidate key from a super key. Every table must have at least one candidate key but at the same time can have several.

In order to be eligible for a candidate key it must pass certain criteria.

* It must contain unique values
* It must not contain null values
* It contains the minimum number of fields to ensure uniqueness
* It must uniquely identify each record in the table
* Once your candidate keys have been identified you can now select one to be your primary key

Primary Key

A primary key is a candidate key that is most appropriate to be the main reference key for the table. As its name suggests, it is the primary key of reference for the table and is used throughout the database to help establish relationships with other tables. As with any candidate key the primary key must contain unique values, must never be null and uniquely identify each record in the table.

**CREATETABLETABLE\_NAME**(

id\_col **INT PRIMARYKEY**,

col2 **CHARACTER VARYING**(20),

...

)

In following example there exists only ONE PRIMARY KEY (pk\_PersonID). However, thevalue of the pk\_PersonID is made up of two columns (P\_Id and LastName).

CREATE TABLE Persons(  
P\_Idint NOT NULL,  
LastNamevarchar(255) NOT NULL,  
FirstNamevarchar(255),  
Address varchar(255),  
City varchar(255),  
CONSTRAINT pk\_PersonID PRIMARY KEY (P\_Id,LastName)  
)

Foreign Key

A foreign key is generally a primary key from one table that appears as a field in another where the first table has a relationship to the second. In other words, if we had a table A with a primary key X that linked to a table B where X was a field in B, then X would be a foreign key in B.

Secondary Key or Alternative Key

A table may have one or more choices for the primary key. Collectively these are known as candidate keys as discuss earlier. One is selected as the primary key. Those not selected are known as secondary keys or alternative keys.

For example in the table showing candidate keys above we identified two candidate keys, studentId and firstName + lastName. The studentId would be the most appropriate for a primary key leaving the other candidate key as secondary or alternative key. It should be noted for the other key to be candidate keys, we are assuming you will never have a person with the same first and last name combination. As this is unlikely we might consider fistName+lastName to be a suspect candidate key as it would be restrictive of the data you might enter. It would seem a shame to not allow John Smith onto a course just because there was already another John Smith.

Simple Key

Any of the keys described before (ie primary, secondary or foreign) may comprise one or more fields, for example if firstName and lastName was our key this would be a key of two fields where as studentId is only one. A simple key consists of a **single field** to uniquely identify a record. In addition the field in itself cannot be broken down into other fields, for example, studentId, which uniquely identifies a particular student, is a single field and therefore is a simple key. No two students would have the same student number.

Compound Key

A compound key consists of more than one field to uniquely identify a record. A compound key is distinguished from a composite key because each field, which makes up the primary key, is also a simple key in its own right. An example might be a table that represents the modules a student is attending. This table has a studentId and a moduleCode as its primary key. Each of the fields that make up the primary key are simple keys because each represents a unique reference when identifying a student in one instance and a module in the other.

Composite

A composite key consists of more than one field to uniquely identify a record. This differs from a compound key in that one or more of the attributes, which make up the key, are not simple keys in their own right. Taking the example from compound key, imagine we identified a student by their firstName + lastName. In our table representing students on modules our primary key would now be firstName + lastName + moduleCode. Because firstName + lastName represent a unique reference to a student, they are not each simple keys, they have to be combined in order to uniquely identify the student. Therefore the key for this table is a composite key.

Unique key

Each unique key is composed from one or more data attributes of that data entity. The set of unique keys declared for a data entity is often referred to as the candidate keys for that data entity.

From the set of candidate keys, a single unique key is selected and declared the primary key for that data entity.

A unique key constraint does not imply the NOT NULL constraint in practice. Because NULL is not an actual value (it represents the lack of a value), when two rows are compared, and both rows have NULL in a column, the column values are not considered to be equal. Thus, in order for a unique key to uniquely identify each row in a table, NULL values must not be used. According to the SQL[3] standard and Relational Model theory, a unique key (unique constraint) should accept NULL in several rows/tuples — however not all RDBMS implement this feature correctly.

A table can have at most one primary key, but more than one unique key.

**CREATE TABLE TABLE\_NAME**(

id\_col **INT**,

col2 **CHARACTERVARYING**(20),

key\_col **SMALL INT**,

...

**CONSTRAINT** key\_unique **UNIQUE**(key\_col),

...

)

CREATE TABLE Persons  
(  
P\_Idint NOT NULL UNIQUE,  
LastNamevarchar(255) NOT NULL,  
FirstNamevarchar(255),  
Address varchar(255),  
City varchar(255)  
)

CREATE TABLE Persons(

P\_Idint NOT NULL,

LastNamevarchar(255) NOT NULL,

FirstNamevarchar(255),

Address varchar(255),

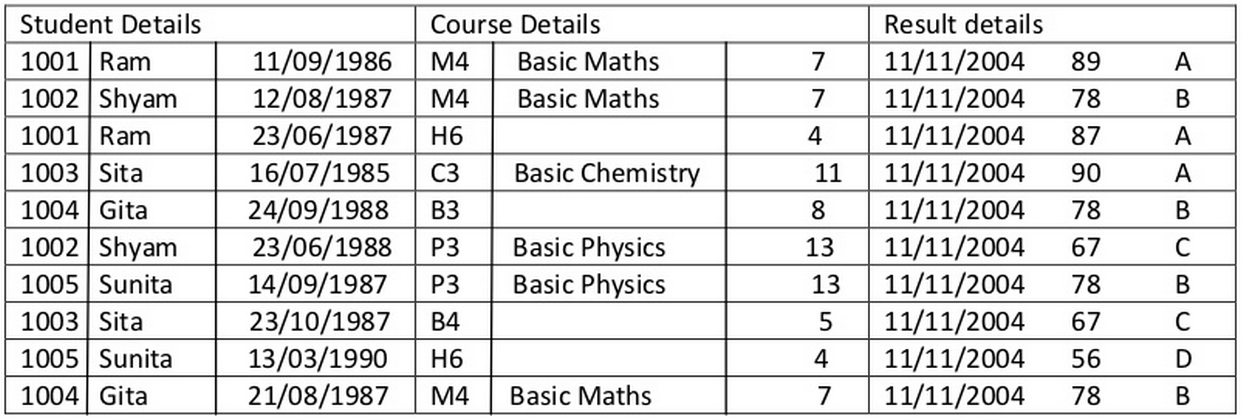
City varchar(255),

CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName)

)

**Normalization**

**{http://www.slideshare.net/jassics/basics-of-normalization-with-examples}**

****

**ER-Model Database**

Let us analyze various anomalies in above good looking table.

* **Insert-anomaly**: We cannot insert any student info without having his course details and vice-versa.
* **update-anomaly**: If we need to update the name of course (say M4) it need to be updated at 3 places.
* **Delete-Anomaly**: If we need to delete any student(Say Ram) , some critical info like Course details will also be deleted. Again it need to be deleted twice.
* **Duplicate-data**: Course detail of Basic-Maths is entered thrice, and Student-details (Ram) is entered twice.This redundancy will increase as database will grow.

**Building-Blocks of Normalization:**

**Determinant**- In result relation Marks determine the grade. i.e. grade is dependent on Marks.

This is represented as Marks->Grade.

To qualify for determinant attribute(Marks) need not be key attribute.

**Functional-Dependency**- Consider following relation schema

REPORT(Student#, Course#, CourseName, InstrucorName, Room#, Marks, Grade)

Here Student#, Course# togather(called composite attribute) defines exactly one value of Marks.

Student#Course#->Marks

Marks is functionaly dependent on Student and course.

Other examples:

Course#->CourseName

Marks->Grade. Etc.

Attribute Y is functionally dependent on X if each value(component) of X determine exactly one value of Y. This is represented as X->Y

X may be composite in nature.

**Full- Functional-Dependency**

In FD, Student#Course#->Marks; Marks is full functionally dependent on Student#Course# because Marks can’t be determined alone either byStudent# or Course#.

Student#Course#->CourseName; CourseName is not full functionally dependent on Student#Course# because student does not play any role on deciding CourseName, It is determined by course# alone.

Attribute Y is fully functionally dependent on X if it is not Functional-Dependent on sub-set of X.

X may be composite in nature.

**Partial-Dependency-**

In FD Student#Course#->CourseName; CourseName is partial dependent on Student#Course#.

Attribute Y is partial dependent on X if it is dependent on sub-set of X.

X may be composite in nature.

**Transitive-Dependency-**

Grade depends on Marks and in turn, marks depends on Student#Course#.

Marks->Grade;

Student#Course#->Marks;

Hence Grade transitively depends on Student#Course#.

Student#Course#->Marks->Grade.

**Key-Attribute –** If the attribute X uniquely defines all other attributes in relationship Rthen X is called key-Attribute which nothing but Candidate-Key.

Student#Course# are key attribute as they uniquely identify other attributes (or whole row).

First Normal-Form[1NF]

A relation is said to be in 1NF iff, all the attributes of the relation R are atomic in nature.

So the above table can be broken into following three tables.

Student-Details=(Student#, Student-name, DOB)

Course-Details=(Course#, Course-name, Prerequisites, Duration)

Results=(Date-of-exam, Marks, Grade)

Second Normal-Form[2NF]

A relation is said to be in 2NF iff,

* It is in 1NF and
* No partial dependency exists between non-key and key attributes.

Student# is key attribute for Student-Details.

Course# is key attribute for Course-Details.

**Joins**

{ http://www.studytonight.com/dbms/joining-in-sql}

SQL Join is used for combining column from two or more tables by using values common to each. A table can also join to itself known as, **Self Join**.

#### Types of Join

#### Cross JOIN or Cartesian Product

It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,

SELECT column-name-list

From *table-name1*

**CROSS JOIN**

*table-name2*;

#### Example:

#### 

**Cross** JOIN query will be,

SELECT \*

from class,

cross JOIN class\_info;

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | Abhi | 1 | DELHI |
| 2 | Adam | 1 | DELHI |
| 4 | Alex | 1 | DELHI |
| 1 | Abhi | 2 | MUMBAI |
| 2 | Adam | 2 | MUMBAI |
| 4 | Alex | 2 | MUMBAI |
| 1 | Abhi | 3 | CHENNAI |
| 2 | Adam | 3 | CHENNAI |
| 4 | Alex | 3 | CHENNAI |

#### INNER Join or EQUI Join

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the query.

Inner Join Syntax is:

SELECT column-name-list

FROM*table-name1***INNER JOIN***table-name2*

WHERE table-name1.column-name = table-name2.column-name;

**Inner** JOIN query will be:

SELECT \*

FROM class, class\_info

WHERE class.id = class\_info.id;

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | Abhi | 1 | DELHI |
| 2 | Adam | 2 | MUMBAI |

#### Natural JOIN

Natural Join is a type of **Inner** **join** which is based on column having same name and same data type present in both the tables to be joined.

Natural Join Syntax is :

SELECT \*

FROM *table-name1* **NATURAL JOIN** *table-name2*;

Example : **Natural join query will be,**

SELECT \*

FROM class NATURAL JOIN class\_info;

The result table will look like,

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **Address** |
| 1 | Abhi | DELHI |
| 2 | Adam | MUMBAI |

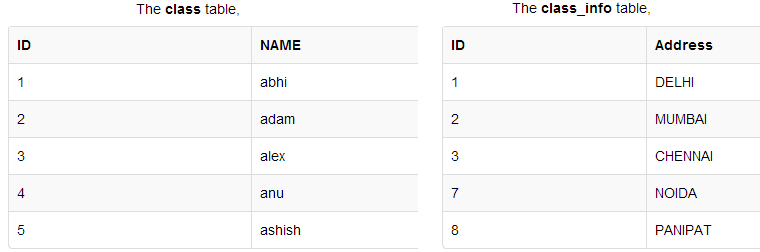
In the above example, both the tables being joined have ID column(same name and same datatype), hence the records for which value of ID matches in both the tables will be the result of Natural Join of these two tables.

#### Outer JOIN

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

* Left Outer Join
* Right Outer Join
* Full Outer Join

Following two table will be used for explaining outer joins.



#### Left Outer Join

Left outer Join Syntax for **Oracle** is,

select column-name-list

from *table-name1*, *table-name2*

on table-name1.column-name = table-name2.column-name(**+**);

**Left Outer Join** query will be,

SELECT \*

FROM class LEFT OUTER JOIN class\_info ON (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | Abhi | 1 | DELHI |
| 2 | Adam | 2 | MUMBAI |
| 3 | Alex | 3 | CHENNAI |
| 4 | Anu | null | Null |
| 5 | ashish | null | Null |

The left outer join returns a result table with the **matched data** of two tables then remaining rows of the **left** table and null for the **right** table's column.

#### Right Outer Join

Right Outer Join Syntax is,

select column-name-list

from *table-name1*

**RIGHT OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

Right outer Join Syntax for **Oracle** is,

select column-name-list

from*table-name1*,

*table-name2*

on table-name1.column-name(**+**) = table-name2.column-name;

**Right Outer Join** query will be,

SELECT \* FROM class RIGHT OUTER JOIN class\_info on (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

The right outer join returns a result table with the **matched data** of two tables then remaining rows of the **right table** and null for the **left** table's columns.

#### Full Outer Join

Full Outer Join Syntax is,

select column-name-list

from*table-name1*

**FULL OUTER JOIN**

*table-name2*

on table-name1.column-name = table-name2.column-name;

**Full Outer Join** query will be like,

SELECT \* FROM class FULL OUTER JOIN class\_info on (class.id=class\_info.id);

The result table will look like,

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **ID** | **Address** |
| 1 | abhi | 1 | DELHI |
| 2 | adam | 2 | MUMBAI |
| 3 | alex | 3 | CHENNAI |
| 4 | Anu | null | null |
| 5 | ashish | null | null |
| null | null | 7 | NOIDA |
| null | null | 8 | PANIPAT |

The full outer join returns a result table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.