SQL Database Introduction to Advance

Database

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just databases.

Data within the most common types of databases in operation today is typically modeled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (SQL) for writing and querying data.

What Is Structured Query Language (SQL)?

SQL is a programming language used by nearly all relational databases to query, manipulate, and define data, and to provide access control. SQL was first developed at IBM in the 1970s with Oracle as a major contributor, which led to implementation of the SQL ANSI standard, SQL has spurred many extensions from companies such as IBM, Oracle, and Microsoft. Although SQL is still widely used today, new programming languages are beginning to appear.

Evolution of the Database

Databases have evolved dramatically since their inception in the early 1960s. Navigational databases such as the hierarchical database (which relied on a tree-like model and allowed only a one-to-many relationship), and the network database (a more flexible model that allowed multiple relationships), were the original systems used to store and manipulate data. Although simple, these early systems were inflexible. In the 1980s, relational databases became popular, followed by object-oriented databases in the 1990s. More recently, NoSQL databases came about as a response to the growth of the internet and the need for faster speed and processing of unstructured data. Today, cloud databases and self-driving databases are breaking new ground when it comes to how data is collected, stored, managed, and utilized.

What’s the Difference Between a Database and a Spreadsheet?

Databases and spreadsheets (such as Microsoft Excel) are both convenient ways to store information. The primary differences between the two are:

- How the data is stored and manipulated
- Who can access the data?
- How much data can be stored

Spreadsheets were originally designed for one user, and their characteristics reflect that. They're great for a single user or small number of users who don't need to do a lot of incredibly complicated data manipulation. Databases, on the other hand, are designed to hold much larger collections of organized information-massive amounts, sometimes. Databases allow multiple users at the same time to quickly and securely access and query the data using highly complex logic and language.

Types of Databases

There are many different types of databases. The best database for a specific organization depends on how the organization intends to use the data.

Relational databases:

Relational databases became dominant in the 1980s. Items in a relational database are organized as a set of tables with columns and rows. Relational database technology provides the most efficient and flexible way to access structured information.

Object-oriented databases:

Information in an object-oriented database is represented in the form of objects, as in object-oriented programming.

Distributed databases:

A distributed database consists of two or more files located in different sites. The database may be stored on multiple computers, located in the same physical location, or scattered over different networks.

Data warehouses:

A central repository for data, a data warehouse is a type of database specifically designed for fast query and analysis.

NoSQL databases:

A NoSQL, or non-relational database, allows unstructured and semi structured data to be stored and manipulated (in contrast to a relational database, which defines how all data inserted into the database must be composed). NoSQL databases grew popular as web applications became more common and more complex.

Graph databases:

A graph database stores data in terms of entities and the relationships between entities.

OLTP databases:

An Online transaction processing database is a speedy, analytic database designed for large numbers of transactions performed by multiple users.

These are only a few of the several dozen types of databases in use today. Other, fewer common databases are tailored to very specific scientific, financial, or other functions. In addition to the different database types, changes in technology development approaches and dramatic advances such as the cloud and automation are propelling databases in entirely new directions. Some of the latest databases include

Open source databases:

An open source database system is one whose source code is open source; such databases could be SQL or NoSQL databases.

Cloud databases:

A cloud database is a collection of data, either structured or unstructured, that resides on a private, public, or hybrid cloud computing platform. There are two types of cloud database models: traditional and database as a service (DBaaS). With DBaaS, administrative tasks and maintenance are performed by a service provider.

Multi Model database:

Multi Model databases combine different types of database models into a single, integrated back end. This means they can accommodate various data types.

Document/JSON database:

Designed for storing, retrieving, and managing document-oriented information, document databases are a modern way to store data in JSON format rather than rows and columns.

Self-driving databases:

The newest and most groundbreaking type of database, self-driving databases (also known as autonomous databases) are cloud-based and use machine learning to automate database tuning, security, backups, updates, and other routine management tasks traditionally performed by database administrators.

Difference between DBMS and RDBMS

Although DBMS and RDBMS both are used to store information in a physical database, there are some remarkable differences between them.

The main differences between DBMS and RDBMS are given below:

No.	DBMS	RDBMS
1	DBMS applications store **data as files**.	RDBMS applications store **data in a tabular form**.
2	In DBMS, data is generally stored in either a hierarchical form or a navigational form.	In RDBMS, the tables have an identifier called primary key and the data values are stored in the form of tables.
3	**Normalization is not** present in DBMS.	**Normalization is** present in RDBMS.
4	DBMS does **not apply any security** with regards to data manipulation.	RDBMS **defines the integrity constraint** for the purpose of ACID (Atomicity, Consistency, Isolation and Durability) property.
5	DBMS uses a file system to store data, so there will be **no relation between the tables**.	In RDBMS, data values are stored in the form of tables, so a **relationship** between these data values will be stored in the form of a table as well.
6	DBMS has to provide some uniform methods to access the stored information.	The RDBMS system supports a tabular structure of the data and a relationship between them to access the stored information.
7	DBMS **does not support distributed databases**.	RDBMS **supports distributed databases**.
8	DBMS is meant to be for small organizations and **deal with small data**. It supports **single users**.	RDBMS is designed to **handle large amounts of data**. it supports **multiple users**.

9	Examples of DBMS are file systems, **xml** etc.	Examples of RDBMS are **mysql**, **postgre**, **sql server**, **oracle** etc.

After observing the differences between DBMS and RDBMS, you can say that RDBMS is an extension of DBMS. There are many software products in the market today which are compatible for both DBMS and RDBMS. Means today a RDBMS application is a DBMS application and vice-versa.

What Is a Database Management System?

A database typically requires a comprehensive database software program known as a database management system (DBMS). A DBMS serves as an interface between the database and its end users or programs, allowing users to retrieve, update, and manage how the information is organized and optimized. A DBMS also facilitates oversight and control of databases, enabling a variety of administrative operations such as performance monitoring, tuning, and backup and recovery.

Some examples of popular database software or DBMSs include MySQL, Microsoft Access, Microsoft SQL Server, FileMaker Pro, Oracle Database, and dBASE.

What Is a MySQL Database?

MySQL is an open source relational database management system based on SQL. It was designed and optimized for web applications and can run on any platform. As new and different requirements emerged with the internet, MySQL became the platform of choice for web developers and web-based applications. Because it's designed to process millions of queries and thousands of transactions, MySQL is a popular choice for ecommerce businesses that need to manage multiple money transfers. On-demand flexibility is the primary feature of MySQL. MySQL is the DBMS behind some of the top websites and web-based applications in the world, including Airbnb, Uber, LinkedIn, Facebook, Twitter, and YouTube.

Using Databases to Improve Business Performance and Decision-Making

With massive data collection from the Internet of Things transforming life and industry across the globe, businesses today have access to more data than ever before. Forward-thinking organizations can now use databases to go beyond basic data storage and transactions to analyze vast quantities of data from multiple systems. Using database and other computing and business intelligence tools, organizations can now leverage the data they collect to run

more efficiently, enable better decision-making, and become more agile and scalable.
The self-driving database is poised to provide a significant boost to these capabilities. Because self-driving databases automate expensive, time-consuming manual processes, they free up business users to become more proactive with their data. By having direct control over the ability to create and use databases, users gain control and autonomy while still maintaining important security standards.

Database Challenges

Today's large enterprise databases often support very complex queries and are expected to deliver nearly instant responses to those queries. As a result, database administrators are constantly called upon to employ a wide variety of methods to help improve performance. Some common challenges that they face include:

Absorbing significant increases in data volume:

The explosion of data coming in from sensors, connected machines, and dozens of other sources keeps database administrators scrambling to manage and organize their companies' data efficiently.

Ensuring data security:

Data breaches are happening everywhere these days, and hackers are getting more inventive. It's more important than ever to ensure that data is secure but also easily accessible to users.

Keeping up with demand:

In today's fast-moving business environment, companies need real-time access to their data to support timely decision-making and to take advantage of new opportunities.

Managing and maintaining the database and infrastructure:

Database administrators must continually watch the database for problems and perform preventative maintenance, as well as apply software upgrades and patches. As databases become more complex and data volumes grow, companies are faced with the expense of hiring additional talent to monitor and tune their databases.

Removing limits on scalability:

A business needs to grow if it's going to survive, and its data management must grow along with it. But it's very difficult for database administrators to predict how much capacity the

company will need, particularly with on-premises databases.
Addressing all of these challenges can be time-consuming and can prevent database administrators from performing more strategic functions.

Purpose of Database Systems:

Database systems arose in response to early methods of computerized management of commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- ✔ Add new students, instructors, and courses
- ✔ Register students for courses and generate class rosters
- ✔ Assign grades to students, compute grade point averages (GPA), and generate transcripts

System programmers wrote these application programs to meet the needs of the university.

Keeping organizational information in a file-processing system has a number of major disadvantages:

Data redundancy and inconsistency.

Since different programmers create the files and application programs over a long period, the various files are likely to have different structures and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files).
For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree. For example, a changed student address may be reflected in the Music department records but not elsewhere in the system.

Difficulty in accessing data:

Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk asks the data-processing department to generate such a list.

Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* students. The university clerk has now two choices: either obtain the list of all students and extract the needed information manually or ask a programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk has the preceding two options, neither of which is satisfactory. The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

Data isolation:

Because data is scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

Integrity problems:

The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

Atomicity problems:

A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $500 from the account balance of department A to the account balance of department *B*. If a system failure occurs during the execution of the program, it is possible that the $500 was removed from the balance of department A but was not credited to the balance of department *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

Concurrent-access anomalies:

For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department *A*, with an account balance of $10,000. If two department clerks debit the account balance (by say $500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department *A* may contain either $9500 or $9900, rather than the correct value of $9400. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously. As another example, suppose a registration program maintains a count of students registered for a course, in order to enforce limits on the number of students registered. When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at (say) 39. The two program executions may both read the value 39, and both would then write back 40,
leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to register, leading to a violation of the limit of 40 students.

Security problems:

Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult. These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems. In most of this book, we use a university organization as a running example of a typical data-processing application.

View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. 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

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

1. **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.
2. **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. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.
3. **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.

Figure 1.1 shows the relationship among the three levels of abstraction. An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programs.

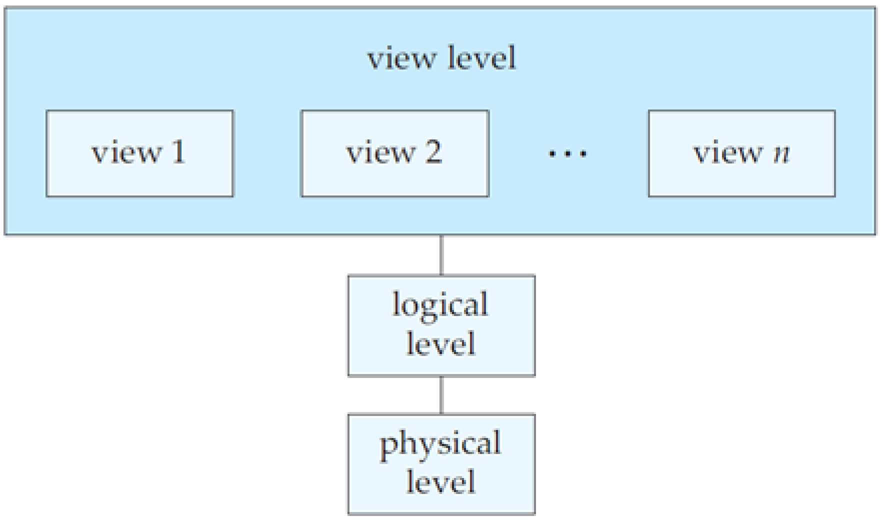

Figure 1.1 The three levels of data abstraction.

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, if at all. The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. 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. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend
on the physical schema, and thus need not be rewritten if the physical schema changes. We study languages for describing schemas after introducing the notion of data models in the next section.

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. A data model provides a way to describe the design of a database at the physical, logical, and view levels. There are a number of different data models that we shall cover in the text. The data models can be classified into four different categories:

1. **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. Tables are also known as **relations**. 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. 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.
2. **Entity-Relationship Model**. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and relationships among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design, and Chapter 7 explores it in detail.
3. **Object-Based Data Model**. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that 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.
4. **Semi structured Data Model**. The semi structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to 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.

Historically, the **network data model** and the **hierarchical data model** preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result, they are used little now, except in old database code that is still in service in some places.

Database Languages

A database system provides a **data-definition language** to specify the database schema and a **data-manipulation language** to express database queries and updates. In practice, the data-definition and data-manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

Data Definition Language (DDL)

DDL is used for specifying the database schema. It is used for creating tables, schema, indexes, constraints etc. in databases. Let’s see the operations that we can perform on database using DDL:

- ❏ To create the database instance – **CREATE**
- ❏ To alter the structure of database – **ALTER**
- ❏ To drop database instances – **DROP**
- ❏ To delete tables in a database instance – **TRUNCATE**
- ❏ To rename database instances – **RENAME**
- ❏ To drop objects from database such as tables – **DROP**
- ❏ To Comment – **Comment**

All of these commands either define or update the database schema that’s why they come under Data Definition language.

Data Manipulation Language (DML)

DML is used for accessing and manipulating data in a database. The following operations on database comes under DML:

- ❏ To read records from table(s) – **SELECT**
- ❏ To insert record(s) into the table(s) – **INSERT**
- ❏ Update the data in table(s) – **UPDATE**
- ❏ Delete all the records from the table – **DELETE**

Data Control language (DCL)

DCL is used for granting and revoking user access on a database –

- ❏ To grant access to user – **GRANT**
- ❏ To revoke access from user – **REVOKE**

In practical data definition language, data manipulation language and data control languages are not separate language, rather they are the parts of a single database language such as SQL.

Transaction Control Language (TCL)

The changes in the database that we made using DML commands are either performed or rollbacked using TCL.

- To persist the changes made by DML commands in database – **COMMIT**
- To roll back the changes made to the database – **ROLLBACK**

Keys:

Key is the unique identity of any entity. Key plays an important role in relational databases; it is used for identifying unique rows from tables. It also establishes relationships among tables.

Primary Key: A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table. The roll number is the primary key because it carries on the unique identity of each student.

Roll	Name	Age
101	Steve	23
102	John	24

Attribute **Name** alone cannot be a primary key as more than one student can have the same name.

Attribute **Age** alone cannot be a primary key as more than one student can have the same age.

Attribute **Roll** alone is a primary key as each student has a unique id that can identify the student record in the table

Super Key: A super key is a set of one or more attributes (columns), which can uniquely identify a row in a table. Often DBMS beginners get confused between super key and candidate key, so we will also discuss candidate key and its relation with super key in this article.

How is the candidate key different from super key?

Answer is simple – Candidate keys are selected from the set of super keys, the only thing we take care while selecting candidate keys is: It should not have any redundant attribute. That's the reason they are also termed as minimal super key. Let’s take an example to understand below table;

Emp_SSN	Emp_Number	Emp_Name
258963	256	Steve
258741	254	Robert

Super keys-The above table has the following super keys. All of the following sets of super keys are able to uniquely identify a row of the employee table.

- {Emp_SSN}
- {Emp_Number}
- {Emp_SSN, Emp_Number}
- {Emp_SSN, Emp_Name}
- {Emp_SSN, Emp_Number, Emp_Name}
- {Emp_Number, Emp_Name}

Candidate Keys- As I mentioned in the beginning, a candidate key is a minimal super key with no redundant attributes. The following two set of super keys are chosen from the above sets as there are no redundant attributes in these sets.

- {Emp_SSN}
- {Emp_Number}

Only these two sets are candidate keys as all other sets are having redundant attributes that are not necessary for unique identification.

Super key vs Candidate Key

I have been getting a lot of comments regarding the confusion between super key and candidate key. Let me give you a clear explanation.

- First you have to understand that all the candidate keys are super keys. This is because the candidate keys are chosen out of the super keys.
- How do we choose candidate keys from the set of super keys? We look for those keys from which we cannot remove any fields. In the above example, we have not chosen {Emp_SSN, Emp_Name} as candidate key because {Emp_SSN} alone can identify a unique row in the table and Emp_Name is redundant.

Primary key-A Primary key is selected from a set of candidate keys. This is done by database admin or database designer. We can say that either {Emp_SSN} or {Emp_Number} can be chosen as a primary key for the table Employee.

Candidate Key: A super key with no redundant attribute is known as candidate key. Candidate keys are selected from the set of super keys, the only thing we take care while selecting candidate key is that the candidate key should not have any redundant attributes. That's the reason they are also termed as minimal super key.

Candidate Key Example

Let's take an example of the table "Employee". This table has three attributes: Emp_Id, Emp_Number & Emp_Name. Here Emp_Id & Emp_Number will have unique values and Emp_Name can have duplicate values as more than one employee can have the same name.

Emp_Id	Emp_Number	Emp_Name
258741	36985214	Steve
369852	25874123	John

How many super keys can the above table have?

- ❏ {Emp_Id}
- ❏ {Emp_Number}
- ❏ {Emp_Id, Emp_Number}
- ❏ {Emp_Id, Emp_Name}
- ❏ {Emp_Id, Emp_Number, Emp_Name}
- ❏ {Emp_Number, Emp_Name}

Let's select the candidate keys from the above set of super keys.

- ❏ {Emp_Id} – No redundant attributes
- ❏ {Emp_Number} – No redundant attributes
- ❏ {Emp_Id, Emp_Number} – Redundant attribute. Either of those attributes can be a minimal super key as both of these columns have unique values.
- ❏ {Emp_Id, Emp_Name} – Redundant attribute Emp_Name.
- ❏ {Emp_Id, Emp_Number, Emp_Name} – Redundant attributes. Emp_Id or Emp_Number alone are sufficient enough to uniquely identify a row of Employee tables.
- ❏ {Emp_Number, Emp_Name} – Redundant attribute Emp_Name.

The **candidate keys** we have selected are:

- ❏ {Emp_Id}
- ❏ {Emp_Number}

Note: A primary key is selected from the set of candidate keys. That means we can either have Emp_Id or Emp_Number as primary keys. The decision is made by DBA (Database administrator)

Alternate Key: Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys. Among these candidate keys, only one key gets selected as the primary key, the remaining keys are known as **alternative or secondary keys**.
Alternate Key Example

Let's take an example to understand the alternate key concept. Here we have a table Employee, this table has three attributes: Emp_Id, Emp_Number & Emp_Name in following table;

Emp_Id	Emp_Number	Emp_Name
258741	325896323	Steve
147852	358694253	John

There are two candidate keys in the above table:
{Emp_Id}
{Emp_Number}
DBA (Database administrator) can choose any of the above keys as primary key. Let's say Emp_Id is chosen as the primary key.
Since we have selected Emp_Id as the primary key, the remaining key Emp_Number would be called alternative or secondary key.

Composite Key: A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.

Note: Any key such as super key, primary key, candidate key etc. can be called composite key if it has more than one attribute.

Composite key Example

Let's consider a table of Sales. This table has four columns (attributes) – cust_Id, order_Id, product_code & product_count in following table;

cust_Id	**order_Id**	**product_code**	**product_count**
125	625	352	35
225	552	356	36

None of these columns **alone** can play a role in this table.

Column **cust_Id** alone cannot become a key as the same customer can place multiple orders, thus the same customer can have multiple entries.

Column **order_Id** alone cannot be a primary key as the same order can contain the order of multiple products, thus the same order_Id can be present multiple times.

Column **product_code** cannot be a primary key as more than one customer can place orders for the same product.

Column **product_count** alone cannot be a primary key because two orders can be placed for the same product count.

Based on this, it is safe to assume that the key should be having more than one attributes:

Key in above table: {cust_id, product_code}

This is a composite key as it is made up of more than one attribute.

Foreign Key: Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables. **For example**;

In the below example the Stu_Id column in the Course_enrollment table is a foreign key as it points to the primary key of the Student table.

Course_enrollment table:

Course_Id	Stu_Id
C01	101
C02	102

Student table:

Stu_Id	Stu_Name	Stu_Age
101	Chaitanya	22
102	Arya	26

Note: Practically, the foreign key has nothing to do with the primary key tag of another table, if it points to a unique column (not necessarily a primary key) of another table then too, it would be a foreign key. So, a correct definition of foreign key would be: Foreign keys are the columns of a table that points to the candidate key of another table.

Constraints:

Constraints enforce limits to the data or type of data that can be inserted/updated/deleted from a table. The whole purpose of constraints is to maintain the **data integrity** during an update/delete/insert into a table. In this tutorial we will learn several types of constraints that can be created in RDBMS.

Types of constraints

NOT NULL

UNIQUE

DEFAULT

CHECK

Key Constraints – PRIMARY KEY, FOREIGN KEY

Domain constraints

Mapping constraints

NOT NULL:

NOT NULL constraint makes sure that a column does not hold NULL value. When we don't provide value for a particular column while inserting a record into a table, it takes NULL value by default. By specifying NULL constraint, we can be sure that a particular column(s) cannot have NULL values.

Example:

```
CREATE TABLE STUDENT(
ROLL_NO INT NOT NULL,
STU_NAME VARCHAR (35) NOT NULL,
STU_AGE INT NOT NULL,
STU_ADDRESS VARCHAR (235),
PRIMARY KEY (ROLL_NO)
);
```

UNIQUE:

UNIQUE Constraint enforces a column or set of columns to have unique values. If a column has a unique constraint, it means that particular column cannot have duplicate values in a table.

```
CREATE TABLE STUDENT(
ROLL_NO INT NOT NULL,
STU_NAME VARCHAR (35) NOT NULL UNIQUE,
STU_AGE INT NOT NULL,
STU_ADDRESS VARCHAR (35) UNIQUE,
PRIMARY KEY (ROLL_NO)
);
```

DEFAULT:

The DEFAULT constraint provides a default value to a column when there is no value provided while inserting a record into a table.

```
CREATE TABLE STUDENT(
ROLL_NO   INT  NOT NULL,
STU_NAME VARCHAR (35) NOT NULL,
STU_AGE INT NOT NULL,
EXAM_FEE INT  DEFAULT 10000,
STU_ADDRESS VARCHAR (35) ,
PRIMARY KEY (ROLL_NO)
);
```

CHECK:

This constraint is used for specifying range of values for a particular column of a table. When this constraint is being set on a column, it ensures that the specified column must have the value falling in the specified range.

```
CREATE TABLE STUDENT(
ROLL_NO   INT  NOT NULL CHECK(ROLL_NO >1000) ,
STU_NAME VARCHAR (35)  NOT NULL,
STU_AGE INT  NOT NULL,
EXAM_FEE INT DEFAULT 10000,
STU_ADDRESS VARCHAR (35) ,
PRIMARY KEY (ROLL_NO)
);
```

In the above example we have set the check constraint on the ROLL_NO column of the STUDENT table. Now, the ROLL_NO field must have a value greater than 1000.

Key constraints:

PRIMARY KEY:

Primary key uniquely identifies each record in a table. It must have unique values and cannot contain nulls. In the below example the ROLL_NO field is marked as primary key, that means the ROLL_NO field cannot have duplicate and null values.

```
CREATE TABLE STUDENT(
ROLL_NO   INT  NOT NULL,
STU_NAME VARCHAR (35)  NOT NULL UNIQUE,
STU_AGE INT NOT NULL,
STU_ADDRESS VARCHAR (35) UNIQUE,
PRIMARY KEY (ROLL_NO)
);
```

FOREIGN KEY:

Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

Domain constraints:

Each table has a certain set of columns and each column allows the same type of data, based on its data type. The column does not accept values of any other data type.

Domain constraints are **user defined data type** and we can define them like this:

Domain Constraint = data type + Constraints (NOT NULL / UNIQUE / PRIMARY KEY / FOREIGN KEY / CHECK / DEFAULT)

Mapping Cardinality:

One to One: An entity of entity-set A can be associated with at most one entity of entity-set B and an entity in entity-set B can be associated with at most one entity of entity-set A.

One to Many: An entity of entity-set A can be associated with any number of entities of entity-set B and an entity in entity-set B can be associated with at most one entity of entity-set A.

Many to One: An entity of entity-set A can be associated with at most one entity of entity-set B and an entity in entity-set B can be associated with any number of entities of entity-set A.

Many to Many: An entity of entity-set A can be associated with any number of entities of entity-set B and an entity in entity-set B can be associated with any number of entities of entity-set A.
We can have these constraints in place while creating tables in the database.

Example:

```
CREATE TABLE Customer (
customer_id int PRIMARY KEY NOT NULL,
first_name varchar(20),
last_name varchar(20)
);

CREATE TABLE Order (
order_id int PRIMARY KEY NOT NULL,
customer_id int,
order_details varchar(50),
constraint fk_Customers foreign key (customer_id)
    references dbo.Customer
);
```

Assuming that a customer orders more than once, the above relation represents **one to many** relations. Similarly, we can achieve other mapping constraints based on the requirements.

Normalization in DBMS: 1NF, 2NF, 3NF and BCNF in Database

Normalization is a process of organizing the data in a database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Let's discuss anomalies first then we will discuss normal forms with examples.

Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let's take an example to understand this.

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id for storing employee's id, emp_name for storing employee's name, emp_address for storing employee's address and emp_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002
123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

Update anomaly: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

Insert anomaly: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.

Delete anomaly: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.

To overcome these anomalies, we need to normalize the data. In the next section we will discuss normalization.

Normalization

Here are the most commonly used normal forms:

- ✔ First normal form(1NF)
- ✔ Second normal form(2NF)
- ✔ Third normal form(3NF)
- ✔ Boyce & Codd normal form (BCNF)

First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212 9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says "each attribute of a table must have atomic (single) values", the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

emp_id	emp_name	emp_address	emp_mobile

101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

✔ Table is in 1NF (First normal form)

✔ No non-prime attribute is dependent on the proper subset of any candidate key of the table.

An attribute that is not part of any candidate key is known as a non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subject, the table can have multiple rows for the same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}

Non-prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non-prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:

teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words, 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as a prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}…so on
Candidate Keys: {emp_id}

Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.
Here, emp_state, emp_city & emp_district depend on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:
employee table:

emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008

1201	Steve	222999

employee_zip table:

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Boyce Codd normal form (BCNF)

It is an advanced version of 3NF that's why it is also referred to as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every functional dependency X->Y, X should be the super key of the table.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

Functional dependencies in the table above:

emp_id -> emp_nationality

emp_dept -> {dept_type, dept_no_of_emp}

Candidate key: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:

emp_nationality table:

emp_id	emp_nationality
1001	Austrian
1002	American

emp_dept table:

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

emp_dept_mapping table:

emp_id	emp_dept
1001	Production and planning
1001	stores
1002	design and technical support
1002	Purchasing department

Functional dependencies:

emp_id -> emp_nationality

emp_dept -> {dept_type, dept_no_of_emp}

Candidate keys:

For first table: emp_id

For second table: emp_dept

For third table: {emp_id, emp_dept}

This is now in BCNF as in both the functional dependencies the left side part is a key.