1/19/2020

Name: Henok Gelaneh

ID: ATR/7217/10

Department: ITSC

Section: IT

Subject: Database Administration

Abstract

A database object is any aspect of a database that you can use to manipulate or hold data. In other words, that covers a wide variety of objects - anything from a saved search to a table could be a database object.



DATABASE OBJECTS

Database Administration

DATABASE OBJECTS

Before we learn about database objects, let's understand -

**What is Data?**

In simple words data can be facts related to any object in consideration. Within a computer's storage, data is a collection of numbers, represented as bytes, that are in turn composed of bits (binary digits) that can have the value one or zero. Data is processed by the CPU, which uses logical operations to produce new data (output) from source data (input).

For example your name, age, height, weight, etc are some data related to you.

A picture , image , file , pdf etc can also be considered data.

**What is a Database?**

Database is a systematic collection of data. Databases support storage and manipulation of data. Databases make data management easy. As defined by Oracle “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 database. “

Let's discuss few examples.

Your electricity service provider is obviously using a database to manage billing , client related issues, to handle fault data, etc.

Let's also consider the facebook and other social media such as telegram. It needs to store, manipulate and present data related to members, their friends, member activities, messages, advertisements and lot more.

We can provide countless number of examples for usage of databases.

**What is a Database Management System (DBMS)?**

Database Management System (also known as DBMS) is a software for storing and retrieving users' data by considering appropriate security measures. It allows users to create their own databases as per their requirement.

It consists of a group of programs which manipulate the database and provide an interface between the database. It includes the user of the database and other application programs.

The DBMS accepts the request for data from an application and instructs the operating system to provide the specific data. In large systems, a DBMS helps users and other third-party software to store and retrieve data.

It also helps to control access to the database.

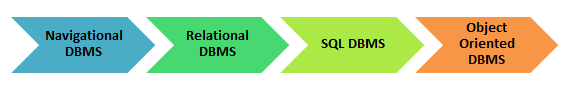
Database Management Systems are not a new concept and as such had been first implemented in 1960s.

Charles Bachmen's Integrated Data Store(IDS) is said to be the first DBMS in history.

With time database technologies evolved a lot while usage and expected functionalities of databases have been increased immensely.

**Types of DBMS**

Let's see how the DBMS family got evolved with the time. Following diagram shows the evolution of DBMS categories.

[](https://www.guru99.com/images/typesofdbms.png)

There are 4 major types of DBMS. Let's look into them in detail.

* **Hierarchical** - this type of DBMS employs the "parent-child" relationship of storing data. This type of DBMS is rarely used nowadays. Its structure is like a tree with nodes representing records and branches representing fields. The windows registry used in Windows XP is an example of a hierarchical database. Configuration settings are stored as tree structures with nodes.
* **Network DBMS** - this type of DBMS supports many-to many relations. This usually results in complex database structures.  RDM Server is an example of a database management system that implements the network model.
* **Relational DBMS** - this type of DBMS defines database relationships in form of tables, also known as relations. Unlike network DBMS, RDBMS does not support many to many relationships. Relational DBMS usually have pre-defined data types that they can support. This is the most popular DBMS type in the market. Examples of relational database management systems include MySQL, Oracle, and Microsoft SQL Server database.
* **Object Oriented Relation DBMS** - this type supports storage of new data types. The data to be stored is in form of objects. The objects to be stored in the database have attributes (i.e. gender, ager) and methods that define what to do with the data. PostgreSQL is an example of an object-oriented relational DBMS.

**The other types of DBMS are**:

* Graph Databases
* ER model Databases
* Document Databases

**Database Objects**

A **database object** is any aspect of a database that you can use to manipulate or hold data. In other words, that covers a wide variety of objects - anything from a saved search to a table could be a database object. In short, almost anything except the data kept in each individual record can be considered a database object. While it may not initially be the easiest concept to fully understand, after this lesson you'll not only be able to explain database objects and identify many of them, but also appreciate just why they are so important to keep organized.

A database object in a relational database is a data structure used to either store or reference data. The most common object that most people interact with is the table. Other objects are indexes, stored procedures, sequences, views and many more.

Oracle distinguishes between *schema objects* and *nonschema objects*.

**Schema Objects**

Database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It’s the database designers who design the schema to help programmers understand the database and make it useful.

A schema is a collection of database objects. A schema is owned by a database user and has the same name as that user. Schema objects are logical structures created by users. Objects such as tables or indexes hold data, or can consist of a definition only, such as a view or synonym.

According to Oracle, Schema objects can be created and manipulated with SQL and include the following types of objects:

* Clusters
* Constraints
* Database links
* Database triggers
* Dimensions
* External procedure libraries
* Index-organized tables
* Indexes
* Index types
* Java classes, Java resources, Java sources
* Materialized views
* Materialized view logs
* Object tables
* Object types
* Object views
* Operators
* Packages
* Sequences
* Stored functions, stored procedures
* Synonyms
* Tables
* Views

**Non-Schema Objects**

Let us first define what a schema-less database is.

1. A schema-less database does not require conformation to a rigid schema (database, schema, data types, tables etc.) that one is required to live up to through the life of a system.
2. Does not enforce data type limitations on individual values pertaining to one single column type
3. Models the business usage and not a database schema, application or product.
4. Can store structured and unstructured data.
5. Eliminates the need to introduce additional layers (ORM layer) to abstract the relational model and expose it in an object-oriented format.

Other types of objects are also stored in the database and can be created and manipulated with SQL but are not contained in a schema:

* Contexts
* Directories
* Parameter files (PFILEs) and server parameter files (SPFILEs)
* Profiles
* Roles
* Rollback segments
* Tablespaces
* Users

**Namespaces**

A namespace defines a group of object types, within which all names must be uniquely identified—by schema and name. Objects in different namespaces can share the same name.

These object types all share the same namespace:

* Tables
* Views
* Sequences
* Private synonyms
* Stand-alone procedures
* Stand-alone stored functions
* Packages
* Materialized views
* User-defined types

So  it is impossible to create a view with the same name as a table; at least, it  
is however impossible if they are in the same schema.

These object types each have their own namespace:

* Indexes
* Constraints
* Clusters
* Database triggers
* Private database links
* Dimensions

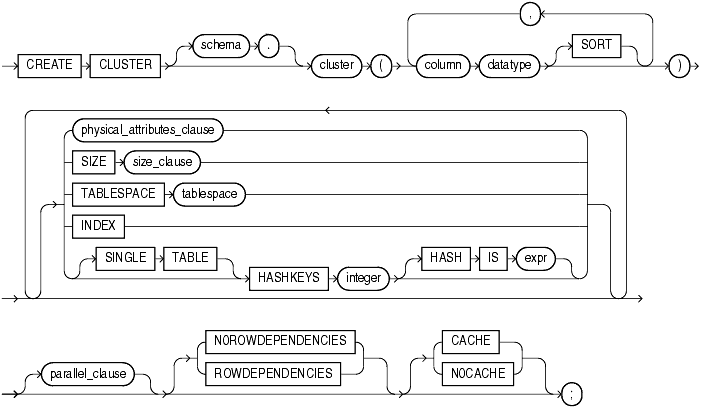
***CLUSTER***

Database Clustering is the process of combining more than one servers or instances connecting a single database. Sometimes one server may not be adequate to manage the amount of data or the number of requests, that is when a Data Cluster is needed. Database clustering, SQL server clustering, and SQL clustering are closely associated with SQL is the language used to manage the database information.

The main reasons for database clustering are its advantages a server receives; Data redundancy, Load balancing, High availability, and lastly, Monitoring and automation.

Database Cluster is highly comprehensive. It covers multiple tiers and arrangements depending on the requirement of the system. Here, we will brief three types of cluster computing architectures. Failover clusters, high-performance clusters, and load balancing clusters.

Despite all these technology distributions at the back-end, it appears to be a single system to the user. The use of clusters varies from enterprise to enterprise, depending on the kind of processes and level of performance required.



***CONSTRAINTS***

Constraints are rules you create at design-time that protect your data from becoming corrupt. It is essential for the long-time survival of your heart child of a database solution. Without constraints your solution will definitely decay with time and heavy usage.

You have to acknowledge that designing your database design is only the birth of your solution. Here after it must live for (hopefully) a long time, and endure all kinds of (strange) behaviour by its end-users (ie. client applications). But this design-phase in development is crucial for the long-time success of your solution! Respect it, and pay it the time and attention it requires.

A wise man once said: *"Data must protect itself!"*. And this is what constraints do. It is rules that keep the data in your database as valid as possible.

There are many ways of doing this, but basically, they boil down to:

* **Foreign key constraints** is probably the most used constraint, and ensures that references to other tables are only allowed if there actually exists a target row to reference. This also makes it impossible to break such a relationship by deleting the referenced row creating a dead link.
* **Check constraints** can ensure that only specific values are allowed in certain column. You could create a constraint only allowing the word 'Yellow' or 'Blue' in a VARCHAR column. All other values would yield an error.
* **Rules** in SQL Server are just reusable Check Constraints (allows you to maintain the syntax from a single place, and making it easier to deploy your constraints to other databases)

The available constraints in SQL are:

* NOT NULL: This constraint tells that we cannot store a null value in a column. That is, if a column is specified as NOT NULL then we will not be able to store null in this particular column any more.
* UNIQUE: This constraint when specified with a column, tells that all the values in the column must be unique. That is, the values in any row of a column must not be repeated.
* PRIMARY KEY: A primary key is a field which can uniquely identify each row in a table. And this constraint is used to specify a field in a table as primary key.
* FOREIGN KEY: A Foreign key is a field which can uniquely identify each row in a another table. And this constraint is used to specify a field as Foreign key.
* CHECK: This constraint helps to validate the values of a column to meet a particular condition. That is, it helps to ensure that the value stored in a column meets a specific condition.
* DEFAULT: This constraint specifies a default value for the column when no value is specified by the user.

As I've hinted here, it takes some thorough considerations to construct the best and most defensive constraint approach for your database design. You first need to know the possibilities and limitations of the different constraint types above.

CREATE TABLE sample table

(

column1 datatype(size) constraint name,

column2 datatype(size) constraint name,

column3 datatype(size) constraint name,

....

);

sample table: Name of the table to be created.

datatype: Type of data that can be stored in the field.

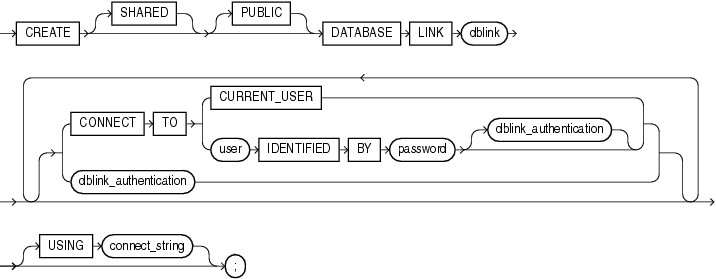
constraint name: Name of the constraint. for example- NOT NULL, UNIQUE, PRIMARY KEY etc.

***DATABASE LINKS***

Use the CREATE DATABASE LINK statement to create a database link. A database link is a schema object in one database that enables you to access objects on another database. The other database need not be an Oracle Database system. However, to access non-Oracle systems you must use Oracle Heterogeneous Services.

After you have created a database link, you can use it to refer to tables and views on the other database. In SQL statements, you can refer to a table or view on the other database by appending @dblink to the table or view name. You can query a table or view on the other database with the SELECT statement. You can also access remote tables and views using any INSERT, UPDATE, DELETE, or LOCK TABLE statement.

To create a private database link, you must have the CREATE DATABASE LINK system privilege. To create a public database link, you must have the CREATE PUBLIC DATABASE LINK system privilege. Also, you must have the CREATE SESSION system privilege on the remote Oracle database.



CREATE DATABASE LINK \_dblink\_name\_

CONNECT TO \_username\_

IDENTIFIED BY \_passwd\_

USING '$\_ORACLE\_SID\_'

***DATABASE TRIGGERS***

A database trigger is special stored procedure that is run when specific actions occur within a database.  Most triggers are defined to run when changes are made to a table’s data.  Triggers can be defined to run *instead of* or *after* DML (Data Manipulation Language) actions such as INSERT, UPDATE, and DELETE.

Triggers help the database designer ensure certain actions, such as maintaining an audit file, are completed regardless of which program or user makes changes to the data.

The programs are called triggers since an event, such as adding a record to a table, fires their execution. The triggers can occur AFTER or INSTEAD OF a DML action.  Triggers are associated with the database DML actions INSERT, UPDATE, and DELETE.  Triggers are defined to run when these actions are executed on a specific table.

create trigger [trigger\_name]

[before | after]

{insert | update | delete}

on [table\_name]

[for each row]

[trigger\_body]

**Explanation of syntax:**

1. create trigger [trigger\_name]: Creates or replaces an existing trigger with the trigger\_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. on [table\_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger\_body]: This provides the operation to be performed as trigger is fired

***DIMENSION***

A database dimension is a collection of related objects, called attributes, which can be used to provide information about fact data in one or more cubes. For example, typical attributes in a product dimension might be product name, product category, product line, product size, and product price. These objects are bound to one or more columns in one or more tables in a data source view. By default, these attributes are visible as attribute hierarchies and can be used to understand the fact data in a cube. Attributes can be organized into user-defined hierarchies that provide navigational paths to assist users when browsing the data in a cube.

Cubes contain all the dimensions on which users base their analyses of fact data. An instance of a database dimension in a cube is called a cube dimension and relates to one or more measure groups in the cube. A database dimension can be used multiple times in a cube. For example, a fact table can have multiple time-related facts, and a separate cube dimension can be defined to assist in analysing each time-related fact. However, only one time-related database dimension needs to exist, which also means that only one time-related relational database table needs to exist to support multiple cube dimensions based on time.

Dimension Designer has three different tabs, which are described in the following table.

|  |  |
| --- | --- |
| Tab | Description |
| Dimension Structure | Use this tab to work with the structure of a dimension-to examine or create the data source view schema for a dimension, to work with attributes, and to organize attributes in user-defined hierarchies. |
| Attribute Relationships | Use this tab to create, modify, or delete the attribute relationships of a dimension. |
| Translations | Use this tab to add and edit translations of dimension metadata in different languages. |
| Browser | Use this tab to examine the members of a processed dimension and to review translations of dimension metadata. |

For example, you can declare a dimension product\_dim, which contains levels product, subcategory, and category:

CREATE DIMENSION products\_dim

LEVEL product IS (products.prod\_id)

LEVEL subcategory IS (products.prod\_subcategory)

LEVEL category IS (products.prod\_category) ...

***INDEX***

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

Indexes are created using a few database columns.

* The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.  
  *Note: The data may or may not be stored in sorted order.*
* The second column is the **Data Reference** or **Pointer** which contains a set of pointers holding the address of the disk block where that particular key value can be found.

The indexing has various attributes:

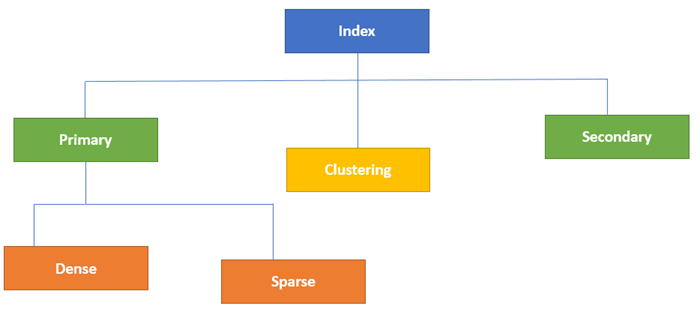
* **Access Types**: This refers to the type of access such as value-based search, range access, etc.
* **Access Time**: It refers to the time needed to find particular data element or set of elements.
* **Insertion Time**: It refers to the time taken to find the appropriate space and insert a new data.
* **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
* **Space Overhead**: It refers to the additional space required by the index.

CREATE INDEX index\_name

ON table\_name (column1, column2, ...);

***INDEX TYPES***

* Clustered
* Multi-dimensional clustered
* Un-clustered, unique,
* non-unique
* b-tree, hash
* GiST
* GIN
* full-text
* bitmap,
* Partitioned
* Function-based



Database Indexing is defined based on its indexing attributes. Two main types of indexing methods are:

* Primary Indexing
* Secondary Indexing

**Primary Indexing**

Primary Index is an ordered file which is fixed length size with two fields. The first field is the same a primary key and second, filed is pointed to that specific data block. In the primary Index, there is always one to one relationship between the entries in the index table.

The primary Indexing is also further divided into two types.

* Dense Index
* Sparse Index

**Dense Index**

In a dense index, a record is created for every search key valued in the database. This helps you to search faster but needs more space to store index records. In this Indexing, method records contain search key value and points to the real record on the disk.

**Sparse Index**

It is an index record that appears for only some of the values in the file. Sparse Index helps you to resolve the issues of dense Indexing. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.

However, sparse Index stores index records for only some search-key values. It needs less space, less maintenance overhead for insertion, and deletions but It is slower compared to the dense Index for locating records.

**Secondary Index**

The secondary Index can be generated by a field which has a unique value for each record, and it should be a candidate key. It is also known as a non-clustering index.

This two-level database indexing technique is used to reduce the mapping size of the first level. For the first level, a large range of numbers is selected because of this; the mapping size always remains small.

**Example of secondary Indexing**

In a bank account database, data is stored sequentially by acc\_no; you may want to find all accounts in of a specific branch of ABC bank.

Here, you can have a secondary index for every search-key. Index record is a record point to a bucket that contains pointers to all the records with their specific search-key value.

**Clustering Index**

In a clustered index, records themselves are stored in the Index and not pointers. Sometimes the Index is created on non-primary key columns which might not be unique for each record. In such a situation, you can group two or more columns to get the unique values and create an index which is called clustered Index. This also helps you to identify the record faster.

**Example:**

Let's assume that a company recruited many employees in various departments. In this case, clustering indexing should be created for all employees who belong to the same dept.

It is considered in a single cluster, and index points point to the cluster as a whole. Here, Department \_no is a non-unique key.

**What is Multilevel Index?**

Multilevel Indexing is created when a primary index does not fit in memory. In this type of indexing method, you can reduce the number of disk accesses to short any record and kept on a disk as a sequential file and create a sparse base on that file.

**B-Tree Index**

B-tree index is the widely used data structures for Indexing. It is a multilevel index format technique which is balanced binary search trees. All leaf nodes of the B tree signify actual data pointers.

Moreover, all leaf nodes are interlinked with a link list, which allows a B tree to support both random and sequential access.

* Lead nodes must have between 2 and 4 values.
* Every path from the root to leaf are mostly on an equal length.
* Non-leaf nodes apart from the root node have between 3 and 5 children nodes.
* Every node which is not a root or a leaf has between n/2] and n children.

**Advantages of Indexing**

Important pros/ advantage of Indexing are:

* It helps you to reduce the total number of I/O operations needed to retrieve that data, so you don't need to access a row in the database from an index structure.
* Offers Faster search and retrieval of data to users.
* Indexing also helps you to reduce tablespace as you don't need to link to a row in a table, as there is no need to store the ROWID in the Index. Thus you will able to reduce the tablespace.
* You can't sort data in the lead nodes as the value of the primary key classifies it.

**Disadvantages of Indexing**

Important drawbacks/cons of Indexing are:

* To perform the indexing database management system, you need a primary key on the table with a unique value.
* You can't perform any other indexes on the Indexed data.
* You are not allowed to partition an index-organized table.
* SQL Indexing Decrease performance in INSERT, DELETE, and UPDATE query.

***MATERIALIZED VIEWS***

**A materialized view** is a view where the query has been executed and the results has been stored as a physical table. You can reference a materialized view in your code much like a real table. In fact, it **is** a real table that you can index, declare constraints etc. When accessing a materialized view, you are accessing the pre-computed results. You are **NOT** executing the underlaying query. There are several strategies for how to keeping the materialized view up-to-date. You will find them all in the documentation.

Materialized views are rarely referenced directly in queries. The point is to let the optimizer use "Query Rewrite" mechanics to internally rewrite a query such as the COUNT(\*) example above to a query on the precomputed table. This is extremely powerful as you don't need to change the original code.

There are many uses for materialied views, but they are mostly used for performance reasons. Other uses are: Replication, complicated constraint checking, workarounds for deficiencies in the optimizer.

-- Normal

CREATE MATERIALIZED VIEW view-name

BUILD [IMMEDIATE | DEFERRED]

REFRESH [FAST | COMPLETE | FORCE ]

ON [COMMIT | DEMAND ]

[[ENABLE | DISABLE] QUERY REWRITE]

AS

SELECT ...;

-- Pre-Built

CREATE MATERIALIZED VIEW view-name

ON PREBUILT TABLE

REFRESH [FAST | COMPLETE | FORCE ]

ON [COMMIT | DEMAND ]

[[ENABLE | DISABLE] QUERY REWRITE]

AS

SELECT ...;

***INDEX ORGANIZED TABLES***

Index Organized Tables (IOT) have their primary key data and non-key column data stored within the same B\*Tree structure. Effectively, the data is stored within the primary key index. There are several reasons to use this type of table.

**Creation Of Index Organized Tables**

To create an index organized table you must:

* Specify the primary key using a column or table constraint.
* Use the ORGANIZATION INDEX.

In addition you can:

* Use PCTTHRESHOLD to define the percentage of the block that is reserved for an IOT row. If the row exceeds this size the key columns (head piece) is stored as normal, but the non-key data (tail piece) is stored in an overflow table. A pointer is stored to locate the tail piece.
* Use OVERFLOW TABLESPACE to define the tablespace that the overflow data will be stored in.
* Use INCLUDING to define which non-key columns are stored with the key columns in the head piece, should overflow be necessary.

CREATE TABLE locations

(id NUMBER(10),

description VARCHAR2(50) NOT NULL,

map BLOB,

CONSTRAINT pk\_locations PRIMARY KEY (id)

)

ORGANIZATION INDEX

TABLESPACE iot\_tablespace

PCTTHRESHOLD 20

INCLUDING description

OVERFLOW TABLESPACE overflow\_tablespace;

***TABLE***

A table comprises of a number of column objects and contains rows of data. A row is a nonempty sequence of values that correspond to the column objects in the table. Every row of the same table has the same number of columns and contains a value for every column of that table.

The following are two types of tables used in the PointBase RDBMS:

* Base Table - a table whose data is actually stored in the database.
* Derived Table - a table obtained from other tables directly or indirectly through the evaluation of a query expression.

A database consists of one or more tables. Each table is made up of rows and columns. If you think of a table as a grid, the column go from left to right across the grid and each entry of data is listed down as a row.

Each row in a relational is uniquely identified by a primary key. This can be by one or more sets of column values. In most scenarios it is a single column, such as employeeID

Every relational table has one primary key. Its purpose is to uniquely identify each row in the database. No two rows can have the same primary key value. The practical result of this is that you can select every single row by just knowing its primary key.

**Columns**

Columns are defined to hold a specific type of data, such as dates, numeric, or textual data.  In the simplest of definitions a column is defined by its name and data type.  The name is used in SQL statements when selecting and ordering data, and the data type is used to validate information stored.

So, a DateOfBirth column defined as a date, can be referred to in an order by clause as

ORDER BY DateOfBirth

And, if you try to add a value of “Hello Kitty” to the column, as part of its validation, it will recognize it isn’t a date, and reject it.

Columns names can’t be duplicated in a table.  So, having two “name” columns is a no no.  Though you could have two “name” columns, such as name1, and name2, you’ll learn later on, that this is frowned up, as it breaks normal form (I explain this in another post).

**Rows**

A table can contain zero or more rows.  When there are zero, it said to be empty.  There is not practical limit on the number of rows a table can hold; however, remember the table’s primary key may have some influence on this.  What I mean, is that if your table holds states, and the primary key is the state’s abbreviation, then by definition, since there are only fifty states in the union, and you can not have duplicates in a primary key, your table is limited to fifty rows.

There is no guarantee that the rows in a table are stored in a particular order.  Use the ORDER BY clause to do so.

Also, strictly speaking, in a relational database there is no first or last row.  Yes, you can tease out a first row of a result using a keyword such as LIMIT or TOP, but those are used once the data is retrieved and sorted.  The difference here is that you’re seeing the first row of the result, not what is physically stored in the table.

CREATE TABLE table\_name (  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
   ....  
);

***OPERATORS***

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. These Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

* Arithmetic operators
* Comparison operators
* Logical operators
* Operators used to negate conditions

**SQL Arithmetic Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** |  |
| + | Add |  |

|  |  |  |
| --- | --- | --- |
| - | Subtract |  |
| \* | Multiply |  |
| / | Divide |  |
| % | Modulo |  |

**SQL Bitwise Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |

**SQL Comparison Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** |  |
| = | Equal to |  |

|  |  |  |
| --- | --- | --- |
| > | Greater than |  |
| < | Less than |  |
| >= | Greater than or equal to |  |
| <= | Less than or equal to |  |
| <> | Not equal to |  |

**SQL Compound Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| += | Add equals |
| -= | Subtract equals |
| \*= | Multiply equals |
| /= | Divide equals |
| %= | Modulo equals |
| &= | Bitwise AND equals |
| ^-= | Bitwise exclusive equals |
| |\*= | Bitwise OR equals |

**SQL Logical Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** |  |
| ALL | TRUE if all of the subquery values meet the condition |  |

|  |  |  |
| --- | --- | --- |
| AND | TRUE if all the conditions separated by AND is TRUE |  |
| ANY | TRUE if any of the subquery values meet the condition |  |
| BETWEEN | TRUE if the operand is within the range of comparisons |  |
| EXISTS | TRUE if the subquery returns one or more records |  |
| IN | TRUE if the operand is equal to one of a list of expressions |  |
| LIKE | TRUE if the operand matches a pattern |  |
| NOT | Displays a record if the condition(s) is NOT TRUE |  |
| OR | TRUE if any of the conditions separated by OR is TRUE |  |
| SOME | TRUE if any of the subquery values meet the condition |  |

***VIEWS***

A database view is a searchable object in a database that is defined by a query.  Though a view doesn’t store data, some refer to a views as “virtual tables,” you can query a view like you can a table.  A view can combine data from two or more table, [using joins](https://www.essentialsql.com/get-ready-to-learn-sql-12-introduction-to-database-joins), and also just contain a subset of information.  This makes them convenient to abstract, or hide, complicated queries.

A view is created from a query using the CREATE VIEW command.  In the example below we are creating a PopularBooks view based of a query which selects all Books that have the IsPopular field checked.  The Query is colored in Blue.

CREATE VIEW PopularBooks AS

SELECT ISBN, Title, Author, PublishDate

FROM Books

WHERE IsPopular = 1

*Database view* is the result set of a query on the data stored in database table(s). Unlike ordinary base tables in a relational database, a view is a virtual table computed or collated dynamically from data in the database when access to that view is requested. Changes applied to the data in a relevant underlying table are reflected in the data shown in subsequent invocations of the view.

Views can provide advantages over tables:

* Views can represent a subset of the data contained in a table.
* Views can join and simplify multiple tables into a single virtual table.

Views take very little space to store; the database contains only the definition of a view, not a copy of all the data that it presents.

You can manipulate nested views, since one view can aggregate data from other views.

The rows of a view are not ordered. View is a relational table, and therefore represents just a set of rows. ORDER BY condition in the view definition is meaningless. However, you can obtain sorted data from a view, just as you do with any other table — as part of a query statement on that view.

***SYNONYMS***

A synonym is an alias for any table, view, materialized view, sequence, procedure, function, package, type, Java class schema object, user-defined object type, or another synonym. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

Synonyms are often used for security and convenience. For example, they can do the following:

* Mask the name and owner of an object
* Provide location transparency for remote objects of a distributed database
* Simplify SQL statements for database users
* Enable restricted access similar to specialized views when exercising fine-grained access control

You can create both public and private synonyms. A public synonym is owned by the special user group named PUBLIC and every user in a database can access it. A private synonym is in the schema of a specific user who has control over its availability to others.

Synonyms are very useful in both distributed and nondistributed database environments because they hide the identity of the underlying object, including its location in a distributed system. This is advantageous because if the underlying object must be renamed or moved, then only the synonym must be redefined. Applications based on the synonym continue to function without modification.

Synonyms can also simplify SQL statements for users in a distributed database system. The following example shows how and why public synonyms are often created by a database administrator to hide the identity of a base table and reduce the complexity of SQL statements. Assume the following:

* A table called SALES\_DATA is in the schema owned by the user JWARD.
* The SELECT privilege for the SALES\_DATA table is granted to PUBLIC.

At this point, you must query the table SALES\_DATA with a SQL statement similar to the following:

SELECT \* FROM jward.sales\_data;

Notice how you must include both the schema that contains the table along with the table name to perform the query.

Assume that the database administrator creates a public synonym with the following SQL statement:

CREATE PUBLIC SYNONYM sales FOR jward.sales\_data;

After the public synonym is created, you can query the table SALES\_DATA with a simple SQL statement:

SELECT \* FROM sales;

Notice that the public synonym SALES hides the name of the table SALES\_DATA and the name of the schema that contains the table.

***SEQUENCES***

A sequence is a database object which allows users to generate unique integer values. The sequence is incremented every time a sequence number is generated. The incrementation occurs even if the transaction rolls back, which may result in gaps between numbers. Similarly, gaps may arise when two users increment the same sequence concurrently.

**Example of Usage**

To create a sequence, the following general statement can be used:

|  |
| --- |
| CREATE SEQUENCE sequence\_name  [MAXVALUE q | NOMAXVALUE]  [MINVALUE v | NOMINVALUE]  [START WITH n]  [INCREMENT BY m]  [CACHE x | NOCACHE]  [CYCLE | NOCYCLE]; |

In the above statement, the MINVALUE and MAXVALUE options define the minimum and the maximum value which the sequence can generate, while the START WITH clause indicates the first sequence number to be generated. It is useful when the sequence is created in an already existing database and some initial values should not be generated. The START WITH value must fall into the range specified by MAXVALUE and MINVALUE.

The INCREMENT BY clause is used to specify the interval between consecutive sequence values. It can be any positive or negative integer different from 0. It is therefore possible to create a sequence object with, for example, decreasing values or even numbers only.

The CACHE option specifies how many values are preallocated and kept in memory for faster access. The larger the cache, the fewer times the database needs to update the root file. On the other hand, greater CACHE values may cause bigger gaps in numbering – if the application makes use of only a few sequence values during the session, the unused numbers will be discarded. The default value for this option varies from one database system to another. In **PostgreSQL**, for example, it equals 1 (i.e. no cache), whereas in **Oracle** it stores 20 sequence numbers by default and the minimum value specified by the user is 2.

The CYCLE clause indicates that the sequence should keep generating numbers after reaching the maximum/minimum value. Note that in such case the next number generated will be MINVALUE or MAXVALUE, respectively, and not the START WITH value. By default, most major database systems apply the NO CYCLE clause and return an error or throw an exception once the limit is reached.

***STORED PROCEDURES***

A stored procedure - or "proc" for short - is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system as a group, so it can be reused and shared by multiple programs. Stored procedures can access or modify data in a database, but it is not tied to a specific database or object. This loose coupling is advantageous because it's easy to reappropriate a proc for a different but similar purpose.

Stored procedures can accept input parameters and return multiple values of output parameters; moreover, stored procedures can program statements to perform operations in the database and return a status value to a calling procedure or batch.

Finally, stored procedures can execute multiple SQL statements, call functions, and even iterate over results sets, performing complex operations akin to programming code. When completed, the proc typically returns one of more result sets to the calling application.

***USER FUNCTIONS***

A function is similar to a stored procedure in that it contains a set of SQL statements that perform a specific task. The idea behind Functions is to foster code reusability. If you have to repeatedly write large SQL scripts to perform the same task, you can create a function that performs that task so that, next time, instead of rewriting the SQL, you can simply call that function. Databases typically include a set of built-in functions that perform a variety of tasks, so always take a look at these before writing your own.

A function accepts inputs in the form of parameters and returns a value. Unlike a stored procedure, a function cannot return a result set. Moreover, functions cannot modify the server environment or operating system environment.

**Main Differences**

While both procs and functions can be employed in similar ways, functions are designed to send their output to a query or SQL statement. Meanwhile, stored procedures are designed to return outputs (i.e. one or more result sets) to the application.

Another difference is that you can group a set of SQL statements and execute them within a stored procedure, stored procedures cannot be called within SQL statements. Functions, on the other hand, may be invoked directly from your queries and/or stored procedures.

Finally, a limitation of functions is that they have to be called for each row. Therefore, if you are using functions with large data sets, you can encounter performance issues.

***DATABASE PACKAGES***

In Oracle, packages encapsulate related procedures, functions, and associated cursors and variables together as a unit in the database. Packages usually have two parts, a specification and a body. The *specification* is the interface with your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The *body* fully defines cursors and subprograms, and so implements the specification.

Packages provide advantages in the following areas:

* *Encapsulation* of related procedures and variables in a single named, stored unit in the database. This provides for better organization during the development process and makes privilege management easier.
* *Separation* of public and private procedures, variables, constants, and cursors.
* Improved *performance* since the entire package is loaded into memory when an object from the package is called for the first time.

You can generate and reverse engineer database packages in the same way as other database objects. When you reverse engineer a database package, the sub-objects (variable, procedure, cursor, exception, and type) are created from the specification and the body of the database package.

**Creating a Database Package (Oracle)**

You can create a database package in any of the following ways:

* Select Model > Database Packages to access the List of Database Packages, and click the Add a Row tool.
* Right-click the model (or a package) in the Browser, and select New > Database Package.

***MATERIALIZED VIEW LOGS***

In an Oracle database, a materialized view log is a table associated with the master table of a materialized view. When master table data undergoes DML changes (such as INSERT, UPDATE, or DELETE), the Oracle database stores rows describing those changes in the materialized view log. A materialized view log is similar to an AUDIT table and is the mechanism used to capture changes made to its related master table. Rows are automatically added to the Materialized View Log table when the master table changes. The Oracle database uses the materialized view log to refresh materialized views based on the master table. This process is called fast refresh and improves performance in the source database.

A materialized view log can capture the primary keys, row IDs, or object identifiers of rows that have been updated in the master table. The standard naming convention for a materialized view log table is: MLOG$\_<master\_name>.

***EXTERNAL PROCEDURE***

An *external procedure* is a third-generation-language routine stored in a dynamic link library (DLL), registered with PL/SQL, and called by you to do special-purpose processing. The routine must be callable from C but can be written in any language.

At run time, PL/SQL loads the library dynamically, then calls the routine as if it were a PL/SQL subprogram. To safeguard your database, the routine runs in a separate address space. But, it participates fully in the current transaction. Furthermore, the routine can call back to the database to do SQL operations.

External procedures promote reusability, efficiency, and modularity. DLLs already written and available in other languages can be called from PL/SQL programs. The DLLs are loaded only when needed, so memory is conserved. Moreover, the DLLs can be enhanced without affecting the calling programs.

Typically, external procedures are used to interface with embedded systems, solve scientific and engineering problems, analyze data, or control real-time devices and processes. For example, you might use external procedures to send instructions to a robot, solve partial differential equations, process signals, analyze time series, or create animation on a video display.

Moreover, external procedures enable you to

* move computation-bound programs from client to server, where they will execute faster thanks to more computing power and less across-network communication
* interface the database server with external systems and data sources
* extend the functionality of the database server itself

If the DBA grants you CREATE ANY LIBRARY privileges, you can create your own alias libraries using the following syntax:

CREATE LIBRARY library\_name {IS | AS} 'file\_path';

**Registering**

EXTERNAL LIBRARY library\_name

[NAME external\_procedure\_name]

[LANGUAGE language\_name]

[CALLING STANDARD {C | PASCAL}]

[WITH CONTEXT]

[PARAMETERS (external\_parameter[, external\_prameter]...)];

where *external\_parameter* stands for

{ CONTEXT

| {parameter\_name | RETURN} [property] [BY REF] [external\_datatype]}

and *property* stands for

{INDICATOR | LENGTH | MAXLEN | CHARSETID | CHARSETFORM}

***PROFILES***

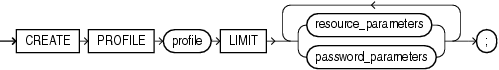
A profile is a collection of parameters that sets limits on database resources. When you change the profile to a user, you assign a profile but you apply also a set of parameters.

If you assign the profile to a user, then that user cannot exceed these limits parameters.

You can use profiles to configure database settings such as:

* sessions per user,
* logging and tracing features,
* controlling user passwords

Use the CREATE PROFILE statement to create a profile, which is a set of limits on database resources. If you assign the profile to a user, then that user cannot exceed these limits.



***DATABASE ROLES***

A database role is a collection of any number of permissions/privileges that can be assigned to one or more users. A database role also is also given a name for that collection of privileges.

The majority of today’s RDBMS’s come with predefined roles that can be assigned to any user. But, a database user can also create his/her own role if he or she has the CREATE ROLE privilege.

**Syntax –**

CREATE ROLE manager;

In the syntax:  
‘manager’ is the name of the role to be created.

* Now that the role is created, the DBA can use the GRANT statement to assign users to the role as well as assign privileges to the role.
* It’s easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user.
* If a role is identified by a password, then GRANT or REVOKE privileges have to be identified by the password.

**Grant privileges to a role –**

GRANT create table, create view

TO manager;

**Grant a role to users**

GRANT manager TO SAM, STARK;

**Revoke privilege from a Role :**

REVOKE create table FROM manager;

**Drop a Role :**

DROP ROLE manager;