## **Data**

Data are values of qualitative or quantitative variables, belonging to a set of items. Simply put, it's an attribute or property or characteristics of an object.

1. **Qualitative Data :**

* **Nominal:** Nominal data represents data where order of the data does not represent any meaningful information**. eg: Passport no.**
* **Ordinal:** Order of the data is important for ordinal data. Eg: Grades, Height – tall, medium, short

1. **Quantitative:**

* **INTERVAL:** Interval classification is used where there is no true zero point in the data and division operation does not make sense. Bank balance, temperature.
* **RATIO**: Ratio class is applied on the data that has a true "zero" and where division does make sense. Consider revenue, length of time.

## **Data Model**

The Data model is specified as a collection of conceptual tools for describing data, data relationships, data semantics and constraints. These models are used to describe the relationship between the entities and their attributes. A data model is a model that describes how data is represented and accessed, usually for a database.

There is the number of data models:

* **Hierarchical data model**: The hierarchical model organizes data into a tree-like structure, where each record has a single parent or root.
* **Network model**: The network model builds on the hierarchical model by allowing many-to-many relationships between linked records, implying multiple parent records.
* **Relational model**: the relational model sorts data into tables, also known as relations, each of which consists of columns and rows
* **Entity-Relationship model**: This model captures the relationships between real-world entities much like the network model, but it isn’t as directly tied to the physical structure of the database.
* **Object-oriented database model**: This model defines a database as a collection of objects, or reusable software elements, with associated features and methods.
* **Document model**
* **Entity-attribute-value model**
* Star schema
* **The object-relational model**: This hybrid database model combines the simplicity of the relational model with some of the advanced functionality of the object-oriented database model.

Points to remember while creating a data model:

* Avoid Large Data Models
* Lack of Clarity or Purpose
* Reckless violation of Normal Form

## **Data Modelling**

Data modeling is the process of creating a data model for the data to be stored in a Database. Data modeling helps in the visual representation of data and enforces business rules, regulatory compliances, and government policies on the data. Data Models ensure consistency in naming conventions, default values, semantics, security while ensuring quality of the data.

Data model emphasizes on what data is needed and how it should be organized instead of what operations need to be performed on the data. Data Model is like an architect's building plan which helps to build a conceptual model and set the relationship between data items.

**Data Modelling** is the diagrammatic representation showing how the entities are related to each other. It is the initial step towards database design. There are three types of data models – **conceptual, logical and physical**. The level of complexity and detail increases from conceptual to logical to a physical data model. We first create the conceptual model, then logical model and finally move to the physical model.

The **conceptual model** shows a very basic high level of design while the **physical data model** shows a very detailed view of design. The conceptual(view) model will be just portraying entity names and entity relationships. The **logical model** will be showing up entity names, entity relationships, attributes, primary keys and foreign keys in each entity. The physical data model will be showing primary keys, foreign keys, table names, column names and column data types. This view actually elaborates how the model will be actually implemented in the database.

## **Dimensional Modeling**

* Dimensional modeling is one of the methods of data modeling that help us store the data in such a way that it is relatively easy to retrieve the data from the database.
* Dimensional modeling represents data with a cube operation, making more suitable logical data representation with OLAP data management.
* Dimensional modeling gives us the advantage of storing data in such a fashion that it is easier to retrieve the information from the data once the data is stored in a database.
* ER Modeling gives us the advantage of storing data in such a way that there is less redundancy.
* The Dimensional model consists of dimension and fact tables. Fact tables store different transactional measurements and the foreign keys from dimension tables that qualify the data.
* **Goals**: Faster Data Retrieval, Better Understandability and Extensibility. The goal of Dimensional model is **not** to achieve high degree of normalization but to facilitate easy and faster data retrieval.
* Ralph Kimball is one of the strongest proponents of this very popular data modeling technique which is often used in many enterprise level data warehouses. The perception of Dimensional Modeling was developed by **Ralph Kimball** and is consist of **"fact"** and **"dimension"** tables
* In the dimensional model, everything is divided in 2 distinct categories - **dimension or measures.**
* **Dimensions** are the object or context. That is - dimensions are the 'things' about which something is being spoken.
* **Measures** are the quantifiable subjects, and these are often numeric.
* In dimensional modeling, the transaction record is divided into either **"facts,"** which are frequently numerical transaction data, or **"dimensions,"** which are the reference information that gives context to the facts. For example, a sale transaction can be damaged into facts such as the number of products ordered and the price paid for the products, and into dimensions such as order date, user name, product number, order ship-to, and bill-to locations, and salesman responsible for receiving the order.

**Step by step approach to dimensional modeling:**

1. Identify the dimensions
2. Identify the measures
3. Identify the attributes or properties of dimensions
4. Identify the granularity of the measures
5. History Preservation (Optional)

**Objectives of Dimensional Modeling**

The purposes of dimensional modeling are:

1. To produce database architecture that is easy for end-clients to understand and write queries.
2. To maximize the efficiency of queries. It achieves these goals by minimizing the number of tables and relationships between them.

**Advantages of Dimensional Modeling**

Following are the benefits of dimensional modeling are:

1. **Dimensional modeling is simple:** Dimensional modeling methods make it possible for warehouse designers to create database schemas that business customers can easily hold and comprehend. There is no need for vast training on how to read diagrams, and there is no complicated relationship between different data elements.
2. **Dimensional modeling promotes data quality:** The star schema enables warehouse administrators to enforce referential integrity checks on the data warehouse. Since the fact information key is a concatenation of the essentials of its associated dimensions, a factual record is actively loaded if the corresponding dimensions records are duly described and also exist in the database. By enforcing foreign key constraints as a form of referential integrity check, data warehouse DBAs add a line of defense against corrupted warehouse data.
3. **Performance optimization is possible through aggregates:** As the size of the data warehouse increases, performance optimization develops into a pressing concern. Customers who have to wait for hours to get a response to a query will quickly become discouraged with the warehouses. Aggregates are one of the easiest methods by which query performance can be optimized.

**Disadvantages of Dimensional Modeling**

1. To maintain the integrity of fact and dimensions, loading the data warehouses with a record from various operational systems is complicated.
2. It is severe to modify the data warehouse operation if the organization adopting the dimensional technique changes the method in which it does business.

**Elements of Dimensional Modeling**

1. **Fact:** It is a collection of associated data items, consisting of measures and context data. It typically represents business items or business transactions.
2. **Dimensions:** It is a collection of data which describe one business dimension. Dimensions decide the contextual background for the facts, and they are the framework over which OLAP is performed.
3. **Measure:** It is a numeric attribute of a fact, representing the performance or behavior of the business relative to the dimensions. Considering the relational context, there are two basic models which are used in dimensional modeling:
   1. Star Model
   2. Snowflake Model

The star model is the underlying structure for a dimensional model. It has one broad central table (fact table) and a set of smaller tables (dimensions) arranged in a radial design around the primary table. The snowflake model is the conclusion of decomposing one or more of the dimensions.

1. **Fact Table:** Fact tables are used to data facts or measures in the business. Facts are the numeric data elements that are of interest to the company.

## **Performance Considerations for Dimensional Modeling**

Performance of a data warehouse is as important as the correctness of data in the data warehouse because unacceptable performance may render the data warehouse as useless. There is this increasing awareness about the fact that it's much more effective to build the performance from the beginning rather than to tune the performance at the end.

Good design, cleanly crafted code and optimal logic/algorithm will give you far better performance than that you can achieve by augmenting your hardware.

1. Surrogate Key is not Mandatory: introduction to surrogate keys in dimension tables necessitates additional lookup operation when doing the fact data loading and lookup is a very costly affair.
2. Keep the data Simple: Do not add extra columns.
3. Reduce Complexity: simplicity will make your model considerably light weight, reducing complexity will make it easier to maintain.
   * Reducing pre-calculated values
   * Removing Unnecessary Constraints
   * Stop snow-flaking
   * Choose the attributes of SCD Type 2 dimensions judiciously
   * Don’t Create a Snapshot Fact and a SCD Type 2 table for the same purpose
   * Consider Indexing Carefully
   * Consider Partitioning
   * Avoid Fact to Fact Join
   * Give some attention to your server infrastructure

## **Levels of data abstraction**

Following are three levels of data abstraction:

**Physical level**: It is the lowest level of abstraction. It describes how data is stored.

**Logical level**: It is the next higher level of abstraction. It describes what data is stored in the database and what the relationship among those data is.

**View level**: It is the highest level of data abstraction. It describes only part of the entire database.

**For example-** User interacts with the system using the GUI and fill the required details, but the user doesn't have any idea how the data is being used. So, the abstraction level is entirely high in VIEW LEVEL.

Then, the next level is for PROGRAMMERS as in this level the fields and records are visible and the programmers have the knowledge of this layer. So, the level of abstraction here is a little low in VIEW LEVEL.

And lastly, the physical level in which storage blocks are described.

## **Common mistakes that are encountered during data modeling**

The common mistakes that are encountered during data modeling activities are listed below:

1. **Building massive data models:** The problem with large massive data models is that they have more design faults. The ideal case scenario is to have a data model build which is under 200 table limits.
2. **Lack of purpose**: Misunderstanding of the business problem, if this is the case then the data model that is built will not suffice the purpose. If you do not know what your business solution is intended for, you might come up with an incorrect data model. So, having clarity on the business purpose is very important to come up with the right data model.
3. **Inappropriate use of surrogate keys**: Surrogate keys should not be used unnecessarily. Use a surrogate key only when the natural key cannot serve the purpose of a primary key.
4. **Unnecessary denormalization**: Don’t denormalize until and unless you have a solid & clear business reason to do so because denormalization creates redundant data which is difficult to maintain.

## **2-Tier & n-Tier** **architecture**

The **2-Tier** **architecture** is the same as basic client-server. In the two-tier architecture, applications on the client end can directly communicate with the database at the server side. Two-tier architecture is the original client-server architecture, consisting of a presentation tier and a data tier; the business logic lives in the presentation tier, the data tier or both. In two-tier architecture the presentation tier - and consequently the end user - has direct access to the data tier, and the business logic is often limited. A simple contact management application, where users can enter and retrieve contact data, is an example of a two-tier application.

**N-tier** **architecture** - also called multi-tier architecture - refers to *any* application architecture with more than one tier. But applications with more than three layers are rare, because additional layers offer few benefits and can make the application slower, harder to manage and more expensive to run. As a result, n-tier architecture and multi-tier architecture are usually synonyms for three-tier architecture.

## **3-tier architecture**

Three-tier architecture, which separates applications into three logical and physical computing tiers, is the predominant software architecture for traditional client-server applications. The 3-Tier architecture contains another layer between the client and server. Introduction of 3-tier architecture is for the ease of the users as it provides the GUI, which makes the system secure and much more accessible. In this architecture, the application on the client-end interacts with an application on the server which further communicates with the database system. A three-tier architecture is a client-server architecture in which the functional process logic, data access, computer data storage and user interface are developed and maintained as independent modules on separate platforms. Three-tier architecture allows any one of the three tiers to be upgraded or replaced independently.

**Benefits**:

* **Faster development**: Because each tier can be developed simultaneously by different teams, an organization can bring the application to market faster, and programmers can use the latest and best languages and tools for each tier. It provides great freedom to development teams who can independently update or replace only specific parts of the application without affecting the product as a whole. Different teams can work on different sections of the application rather than on the full stack according to their areas of expertise, improving their efficiency and speed. Development cycle or upgrade times are significantly improved ensuring minimal disruption in customer’s experience.
* **Improved scalability**: Any tier can be scaled independently of the others as needed. The application can be scaled up and out rather easily by detaching the front-end application from the databases that are selected according to the individual needs of the customer. New hardware, such as new servers, can also be added at a later time to deal with massive amounts of data or particularly demanding services. A three-tier-architecture also provides a higher degree of flexibility to enterprises who may want to adopt a new technology as soon as it becomes available.
* **Improved reliability**: An outage in one tier is less likely to impact the availability or performance of the other tiers.
* **Improved security**: Because the presentation tier and data tier can't communicate directly, a well-designed application tier can function as a sort of internal firewall, preventing SQL injections and other malicious exploits. Critical components of the application can be encapsulated and retained while the whole system keeps evolving organically.

In web development, the tiers have different names but perform similar functions:

* The **web server**is the presentation tier and provides the user interface. This is usually a web page or web site, such as an ecommerce site where the user adds products to the shopping cart, adds payment details or creates an account. The content can be static or dynamic, and is usually developed using HTML, CSS and Javascript .
* The **application server**corresponds to the middle tier, housing the business logic used to process user inputs. To continue the ecommerce example, this is the tier that queries the inventory database to return product availability, or adds details to a customer's profile. This layer is often developed using Python, Ruby or PHP and runs a framework such as e Django, Rails, Symphony or ASP.NET, for example.
* The **database server**is the data or backend tier of a web application. It runs on database management software, such as MySQL, Oracle, DB2 or PostgreSQL, for example.

## **Database**

A [database](https://www.techopedia.com/definition/1185/database-db) is a structured assortment of related data. It is processed, organized, managed and updated, then stored electronically. It’s a popular method used by organizations to **store information that needs to be retrieved frequently**.

Main Characteristics of a Database:

* Organized according to company operations and applications
* Highly structured
* Fast retrieval and understandable system
* [OLTP](https://database.guide/what-is-oltp/) (online transaction processing) application
* Data recording capabilities

## **Data mart**

Data marts are generally designed for a single subject area. An organization may have data pertaining to different departments like Finance, HR, Marketing etc. stored in a data warehouse and each department may have separate data marts. These **data marts can be built on top of the data warehouse**. The **data mart is a subset of the data warehouse** and is usually oriented to a specific business line or team.

A [data mart](https://www.dataversity.net/difference-data-mart-data-warehouse/) is a preferred method when working with departmental data because a data mart is a repository for summarized data derived from the data warehouse. The data mart offers **subject-oriented data** that benefits a specific set of people within the organization. For example, the company executives or the sales team might use a data mart for marketing analysis. An enterprise would want to leverage a data mart vs. a data warehouse. Primarily because a data mart is smaller in scope, focusing on a single area.

Main Characteristics of a Data Mart:

* Focuses on one subject matter
* Dedicated to only one business function
* Only stores one subset of data
* Often uses a star schema or similar structure

## **Data Lake**

A [data lake](https://www.forbes.com/sites/bernardmarr/2018/08/27/what-is-a-data-lake-a-super-simple-explanation-for-anyone/) stores an organization’s raw and processed data at both large and small scales. Different users in the organization can dive in and retrieve the relevant data for their department to use. Data lakes store raw data and are more flexible but less secure, and often need data scientists to understand them. While similar in bandwidth and both possessing the ability to store large amounts of data, a data lake vs. a data warehouse differentiate in the types of data they store.

**Main Characteristics of a Data Lake:**

* Collects all data from various sources over an extended period of time
* Meets the needs of various users in the organization
* Is uploaded without an established methodology

**Best practices for Data Lake Implementation:**

* Architectural components, their interaction and identified products should support native data types.
* Design of Data Lake should be driven by what is available instead of what is required. The schema and data requirement is not defined until it is queried.
* Design should be guided by disposable components integrated with service API.
* Data discovery, ingestion, storage, administration, quality, transformation, and visualization should be managed independently.
* The Data Lake architecture should be tailored to a specific industry. It should ensure that capabilities necessary for that domain are an inherent part of the design
* Faster on-boarding of newly discovered data sources is important
* Data Lake helps customized management to extract maximum value
* The Data Lake should support existing enterprise data management techniques and methods

**Challenges of building a data lake:**

* In Data Lake, Data volume is higher, so the process must be more reliant on programmatic administration.
* It is difficult to deal with sparse, incomplete, volatile data.
* Wider scope of dataset and source needs larger data governance & support.

## **Data Warehouse**

* A data warehouse is an electronic storage of an Organization's historical data for the purpose of Data Analytics, such as reporting, analysis, and other knowledge discovery activities. It deals with the mechanism of electronically storing and retrieving data so that some analysis can be performed on that data to corroborate / support a business decision or to predict a business outcome.
* A data warehouse is a subject oriented, non-volatile, integrated, time variant collection of data in support of management's decisions. It helps to integrate data and store them historically so that we can analyze different aspects of business including, performance analysis, trend, prediction etc. over a given time frame and use the result of our analysis to improve the efficiency of business processes.
* A data warehouse is a repository of an organization's electronically stored data. Data warehouses are designed to facilitate reporting and analysis.
* Data warehousing is the science of storing data for the purpose of meaningful future analysis.
* Other than Data Analytics, a data warehouse can also be used for the purpose of data integration, master data management etc.

Subject Oriented This means a data warehouse has a defined scope and it only stores data under that scope. So, for example, if the sales team of your company is creating a data warehouse - the data warehouse by definition is required to contain data related to sales (and not the data related to production management)

Non-volatile This means that data once stored in the data warehouse are not removed or deleted from it and always stay there no matter what. data once loaded in the warehouse will not get deleted later.

Integrated This means that the data stored in a data warehouse make sense. Fact and figures are related to each other and they are integrable and project a single point of truth.

Time variant This means that data is not constant, as new and new data gets loaded in the warehouse, data warehouse also grows in size which changes wrt time.

## **Benefits of data warehouse**

* A data warehouse helps to integrate data and store it historically so that we can analyze different aspects of business including, performance analysis, trend, prediction etc. over a given time frame and use the result of our analysis to improve the efficiency of business processes.
* DW technologies provide historical, current and predictive views of business operations by analyzing the present and historical business data. Data analysis is often done using visualization techniques that turn complex data into images that tell a compelling story. Raw data by this process of analysis helps management make right decisions.
* For a long time in the past and also even today, Data warehouses are built to facilitate reporting on different key business processes of an organization, known as KPI (Key Performance Index). Today we often call this whole process of reporting data from data warehouses as "Data Analytics". Data warehouses also help to integrate data from different sources and show a single-point-of-truth values about the business measures (e.g. enabling Master Data Management).
* Data warehouse can be further used for data mining which helps trend prediction, forecasts, pattern recognition etc.



## **What issues should be taken into consideration during the database design?**

**Semantics of Attributes:** The relation schema should be designed in such a way that it should have a clear meaning. Combining attributes from various entity types and relation types into a single relation should be avoided.

**Redundancy:** Information should be stored only once to minimize the wastage of storage space else the database size would keep on increasing. Redundancy might arise if two or more relations are kept in a single Database Relation.

**NULL values:** Attributes which have frequent NULL values should be avoided in base relations as NULL values might cause following problems:

1. Multiple interpretations might corrupt the semantics of relation
2. This might cause issues in aggregate operations like SUM, COUNT, etc.

**Non-Additive join(lossless) property:** While joining relations based on equality conditions on attributes which might be either primary keys or foreign keys, it is guaranteed that no spurious tuples would be generated.

* Redundancy should be avoided. Information should be stored only once to save space and other relation problems
* Each cell should have atomic values according to 1NF
* Relationships between tables should have clear meanings.
* NULL values should be avoided as it may corrupt the semantics of relation and cause issues in aggregate operations.

## **Advantages of DBMS over Traditional file-based system**

Database management systems were developed to handle the following difficulties of typical File-processing systems supported by conventional operating systems.

1. Data redundancy and inconsistency
2. Difficulty in accessing data
3. Data isolation – multiple files and formats
4. Data Integrity problems
5. Atomicity of updates
6. Concurrent access by multiple users: Concurrent access is not possible in file based
7. Security problems: file based not secure
8. Limited data sharing

## **How do Industries utilize databases?**

[Industries that use databases](https://smallbusiness.chron.com/database-uses-business-64298.html) need to have a highly efficient system of data retrieval for smooth operations. As technology and ecommerce expands, databases are a ubiquitous data processing tool for most industries. But these industries, in particular, rely heavily on databases.

**Airline:** The airline database generates important reports like the flight manifest, and it’s also used for scheduling flights and creating passengers’ reservations.

**Telecommunication:** From their database, a telecommunication company generates customer bills, call logs, balances for pre-paid customers among other crucial operational information.

**Sales:** The sales department of any organization is perhaps the biggest beneficiary of the company’s database. The system enables them to track sales, customer information and product performance.

**Banking:** The banking sector relies heavily on databases to process their transactions and maintain up-to-date customer information and details. A properly updated database is also crucial to accuracy in serving customers.

## **How do Industries utilize data warehouses?**

[Data warehousing applies to industries](http://tdan.com/data-warehouse-applications-by-industry/5070) that have a large volume of data to processes frequently. They include healthcare and insurance, as well as finance, government, education, services, and manufacturing. With heightened security, data sensitive industries prefer data warehouses vs. databases.

**Healthcare**: The healthcare sector has a lot of information being inputted on a daily basis from stakeholders to suppliers and of course, patients. This data is organized and stored in the warehouse, and can later be accessed to create treatment plans, strategize on purchases and processes and even predict epidemics in advance.

**Insurance**: Insurance is another sector that sees a huge, continuous flow of data. Using a data warehouse allows the industry stakeholders to have current information on customer patterns and create a quick analysis of market trends. Because insurance is always changing, a quick way to share data is crucial to keep up with the industry changes.

## **How do Industries utilize data lakes?**

A [data lake](https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/a-smarter-way-to-jump-into-data-lakes) is an excellent, complementary tool to a data warehouse because it provides more query options. A data warehouse will provide structured and organized information. However, with the addition of a data lake the organization can tap into raw data that may offer even more insight or support because data lakes **provide real-time analytics**. A data mart vs. data lake creates two sides of the spectrum, where data marts are focused data and data lakes are huge repositories of raw data.

**Research and Science:** Science is ever evolving, and it relies on real time data to make crucial deductions. Data lakes are suitable for scientific use because not only is the data raw from feedback sources and algorithms, it’s also real time. Science is only as good as its most current and relevant deductions. **Research needs to be fresh** to have an impact on the reports or findings that it produces.

**IT:** IT architects can access data from the data lake in its most original form and scale it up or down depending on their needs. By using raw data, the organization is able to **create more accurate products** that cater better to customer needs.

## **Master Data Management (MDM)**

Master data management (MDM) is the core process used to manage, centralize, organize, categorize, localize, synchronize and enrich master data according to the business rules of the sales, marketing and operational strategies of your company. **Master data management (MDM)** is a technology-enabled discipline in which business and IT work together to ensure the uniformity, accuracy, stewardship, semantic consistency and accountability of the enterprise’s official shared master data assets. Master data is the consistent and uniform set of identifiers and extended attributes that describes the core entities of the enterprise including customers, prospects, citizens, suppliers, sites, hierarchies and chart of accounts.

Master data can take the form of product, customer, supplier, location and asset information, in addition to any information sources that drive your business.

The efficient management of master data in a central repository gives you a single authoritative view of information and eliminates costly inefficiencies caused by data silos. Master data is used by multiple applications, an error in the data in one place can cause errors in all the applications that use it. It supports your business initiatives and objectives through identification, linking and syndication of information and content across products, customers, stores/locations, employees, suppliers, digital assets and more.

As businesses grow, their IT infrastructure gets more complicated. Acquisitions and expansion into new markets and countries can result in multiple systems, applications and technologies. This fragmented environment creates significant data management difficulties.

MDM provides an organized approach to data management and increases the value and quality of data. It connects information, people and processes to help businesses achieve objectives and improve results. The key to making master data both trustworthy and insightful is transparency, delivered by an accountable organization which specifies data’s meaning, purpose and governance policy.

MDM defines and implements these policies to certify data’s origin, accuracy, coherence, accessibility, security, auditability and ethics are under supervision and best serving business purpose.

There are two basic steps to creating master data:

1. Cleaning and standardizing the data
2. Matching data from all the sources to consolidate duplicates.

MDM connects, masters and shares data from all your systems, including ERP, CRM, ecommerce and more. It allows you to create a 360° view of your information, including everything from the buying history of your customers to product availability and supplier interaction.

Master data management allows businesses to:

* Focus product, service and business efforts on activities that increase sales, and de-prioritize those draining time and resources.
* Accelerate data maturity by improving the accuracy, visibility and clarity required in digital business.
* Enable data transparency to meet the evolving needs of customers for personalized, engaging data-rich experiences.

## **RDBMS**

RDBMS (Relational Database Management System) is an information management system, which is based on a data model. In RDBMS tables are used for information storage. Each row of the table represents a record and a column represents an attribute of data. Organization of data and their manipulation processes are different in RDBMS from other databases. RDBMS ensures ACID (atomicity, consistency, integrity, durability) properties required for designing a database. The purpose of RDBMS is to store, manage, and retrieve data as quickly and reliably as possible.

## **OLTP**

Online transaction processing provides transaction-oriented applications in a 3-tier architecture. OLTP administers day to day transactions of an organization and mainly deals with large numbers of transactions by a large number of users. OLTP is known for maintaining transactional level data of the organization and business and is generally highly normalized. OLTP (Online Transactional Processing) is a category of data processing that is focused on transaction-oriented tasks. It typically involves inserting, updating, and/or deleting small amounts of data in a database.

OLTP transactions are usually very specific in the task that they perform, and they usually involve a single record or a small selection of records. If it is an OLTP route then it is going to be a **star schema** design. OLTP handles the ACID properties during data transactions via the application.

**Examples –** Uses of OLTP are as follows:

* ATM center
* Online banking
* Purchasing a book online
* Online airline ticket booking
* Sending a text message
* Order entry
* Telemarketers entering telephone survey results
* Call center staff viewing and updating customers’ details

OLTP applications typically possess the following characteristics:

* Transactions that involve small amounts of data
* Indexed access to data
* A large number of users
* Frequent queries and updates
* Fast response times

## **OLAP**

Online Analytical Processing consists of a type of software tools that are used for data analysis for business decisions. OLAP provides an environment to get insights from the database retrieved from multiple database systems at one time. OLAP is for analysis and reporting purposes & it is in de-normalized form. **OLAP (Online Analytical Processing)** is the technology behind many [Business Intelligence (BI)](http://olap.com/business-intelligence-systems/)applications. OLAP is a powerful technology for data discovery, including capabilities for limitless report viewing, complex analytical calculations, and predictive “what if” scenario (budget, forecast) planning.

OLAP performs **multidimensional analysis** of business data and provides the capability for complex calculations, trend analysis, and sophisticated data modeling. It is the foundation for many kinds of business applications for Business Performance Management, Planning, Budgeting, Forecasting, Financial Reporting, Analysis, Simulation Models, Knowledge Discovery, and Data Warehouse Reporting. OLAP enables end-users to perform ad hoc analysis of data in multiple dimensions, thereby providing the insight and understanding they need for better decision making.

Unlike relational databases, OLAP tools do not store individual transaction records in two-dimensional, row-by-column format, like a worksheet, but instead use multidimensional database structures known as **Cubes** to store arrays of consolidated information. The cubes are designed in such a way that creating and viewing reports become easy. The data and formulas are stored in an optimized multidimensional database, while views of the data are created on demand. If it is an OLAP route then it is going to be a **snowflake** schema design.

**Examples –** Any type of Data warehouse system is an OLAP system. Uses of OLAP are as follows:

* Spottily analyzed songs by users to come up with the personalized homepage of their songs and playlist.
* Netflix movie recommendation system.

An OLAP solution might be intended for dynamic reporting for finance professionals, with source data originating in an ERP system. Or a solution might address a medical institution’s activities as concerns patient analysis. All of which is to say that customers need to have clear objectives in mind for an intended solution, and start to consider product selection on that basis.

**OLAP** is a technology that enables analysts to extract and view business data from different points of view. Analysts frequently need to group, aggregate and join data. These operations in relational databases are resource intensive. With OLAP data can be pre-calculated and pre-aggregated, making analysis faster. OLAP technology has been defined as the ability to achieve “fast access to shared multidimensional information.”

Basic analytical operations of OLAP. Four types of analytical operations in OLAP are:

* Roll-up
* Drill-down
* Slice and dice
* Pivot (rotate)

## **Types of OLAP**

| **Type of OLAP** | **Explanation** |
| --- | --- |
| **Relational OLAP(ROLAP):** | ROLAP is an extended RDBMS along with multidimensional data mapping to perform the standard relational operation. |
| **Multidimensional OLAP (MOLAP)** | MOLAP implements operations in multidimensional data. |
| **Hybrid Online Analytical Processing (HOLAP)** | In HOLAP approach the aggregated totals are stored in a multidimensional database while the detailed data is stored in the relational database. This offers both data efficiency of the ROLAP model and the performance of the MOLAP model. |
| **Desktop OLAP (DOLAP)** | In Desktop OLAP, a user downloads a part of the data from the database locally, or on their desktop and analyze it. DOLAP is relatively cheaper to deploy as it offers very few functionalities compared to other OLAP systems. |
| **Web OLAP (WOLAP)** | Web OLAP which is an OLAP system accessible via the web browser. WOLAP is a three-tiered architecture. It consists of three components: client, middleware, and a database server. |
| **Mobile OLAP:** | Mobile OLAP helps users to access and analyze OLAP data using their mobile devices |
| **Spatial OLAP :** | SOLAP is created to facilitate management of both spatial and non-spatial data in a Geographic Information system (GIS) |

## **ER model**

ER model or entity-relationship model is a particular methodology of data modeling wherein the goal of modeling is to normalize the data by reducing redundancy. This is different from **dimensional modeling** where the main goal is to improve the data retrieval mechanism. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.

It develops a conceptual design for the database. It also develops a very simple and easy to design view of data. In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.

Entity Relationship Diagram, also known as ERD, ER Diagram or ER model, is a type of structural diagram for use in database design. An ERD contains different symbols and connectors that visualize two important information: The major entities within the system scope, and the inter-relationships among these entities.

Businesses with complex databases – in other words, lots of data – might use ERD. It's mainly used by analysts when designing a database. The database helps them communicate the landscape of the business to different teams, and this overview will help you build the applications needed to support the business.

ER diagrams are created based on three basic concepts: entities, attributes and relationships. ER Diagrams contain different symbols that use rectangles to represent entities, ovals to define attributes and diamond shapes to represent relationships. This model is based on the real world. It contains necessary objects (known as entities) and the relationship among these objects.

In ER diagram, entities are represented by rectangles, relationships are represented by diamonds, attributes are the characteristics of entities and represented by ellipses, and data flow is represented through a straight line.

The **Entity** is a set of attributes in a database. An entity can be a real-world object which physically exists in this world. All the entities have their attributes which in the real world are considered as the characteristics of the object. For example: In the employee database of a company, the employee, department, and the designation can be considered as the entities. These entities have some characteristics which will be the attributes of the corresponding entity.

An entity type is specified as a collection of entities, having the same attributes. Entity type typically corresponds to one or several related tables in the database. Eg: a student has student\_id, department, and course as its characteristics.

The entity set specifies the collection of all entities of a particular entity type in the database. An entity set is known as the set of all the entities which share the same properties. Eg: a set of people, a set of students, a set of companies.

An entity set that doesn't have sufficient attributes to form a primary key is referred to as a weak entity set. The member of a weak entity set is known as a subordinate entity.

## **Attribute**

An attribute refers to a database component. It is used to describe the property of an entity. An attribute can be defined as the characteristics of the entity. Entities can be uniquely identified using the attributes. Attributes represent the instances in the row of the database. For example: If a student is an entity in the table then age will be the attribute of that student.

## **Dimension**

A dimension is something that qualifies a quantity (measure). Dimensions are mutually independent. Technically speaking, a dimension is a data element that categorizes each item in a data set into non-overlapping regions. Dimensions are the object or context. That is - dimensions are the 'things' about which something is being spoken.

For an example, consider this: If I just say… “20kg”, it does not mean anything. But if I say, "20kg of Rice (Product) is sold to Ramesh (customer) on 5th April (date)", then that gives a meaningful sense. These product, customer and dates are some dimension that qualified the measure - 20kg.

## **Fact**

A fact is something that is quantifiable (or measurable). Facts are typically (but not always) numerical values that can be aggregated.

**Measures** are the quantifiable subjects and these are often numeric which are stored in a separate table called Fact table.

## **Types of Fact table**

The fact table is a central table in the data schemas. It is found in the center of a star schema or snowflake schema and surrounded by a dimension table. It contains the facts of a particular business process, such as sales revenue by month. Facts are known as measurements or matrices. It captures a measurement or a metric. It is an essential concept for data warehousing and BI Certification. The fact table stores quantitative information of analysis that is not arranged. The fact table is a primary table in the dimensional model. It also contains measurement, metric and quantitative information.

There are three types of facts:

1. **Summative facts**: Summative facts are used with aggregation functions such as sum (), average (), etc.
2. **Semi summative facts**: There are small numbers of quasi-summative fact aggregation functions that will apply.  
   For example, consider bank account details. We also cannot also apply () for a bank balance which will not have useful results, but the minimum() and maximum() functions return useful information.
3. **Non-additive facts**: We cannot use numerical aggregation functions such as sum (), average (), on non-additive facts. For non-additive facts, ratio or percentage is used.

There are three types of fact tables:

1. **Transaction Fact Table**

The transaction fact table is a basic approach to operate the businesses. These fact tables represent an event that occurs at the primary point. A line exists in the fact table for the customer or product when the transaction occurs.

Many rows in a fact table connect to a customer or product because they are involved in multiple transactions. Transaction data is often structured quickly in a one-dimensional framework. The lowest-level data is the rawest dimensional data that cannot be done by summarized data.

1. **Snapshot Fact Table**

The snapshot fact table describes the state of things at a particular time and contains many **semi-additive** and **non-additive** facts.

**Example:** The daily equilibrium fact is expressed by the customer dimension but not by the time dimension.

Periodic snapshots require the performance of the business at regular and estimated time intervals. Unlike a transaction fact table where we load a row for each event, with periodic snapshots, we take a picture of the activity at the end of the day, week, or month, and then another picture at the end of the next period.

**Example:** Performance summary of a salesman during the previous month.

1. **Accumulated Fact Sheet**

The accumulated fact table is used to show the activity of a process that has a beginning and an end.

For example, we are processing an order. An order remains in the process until it will be processed. As the step towards completing the order is completed, the corresponding row in the fact table is updated.

## **Types of Measures**

**Non-additive measures:**

Non-additive measures are those which cannot be used inside any numeric aggregation function (e.g. SUM(), AVG() etc.). One example of non-additive fact is any kind of ratio or percentage column; a flag or an indicator column present in fact table holding values like Y/N, etc. is a non-additive measure. Example, 5% profit margin, revenue to asset ratio etc. A non-numerical data can also be a non-additive measure when that data is stored in fact tables, e.g. some kind of varchar flags in the fact table.

**Semi-additive measures:**

Semi-additive measures are those where only a subset of aggregation function can be applied. Let’s say account balance. A sum() function on balance does not give a useful result but max() or min() balance might be useful. Consider price rate or currency rate. Sum is meaningless on rate; however, average function might be useful.

**Additive measures:**

Additive measures can be used with any aggregation function like Sum(), Avg() etc. Example is Sales Quantity, units purchased etc.

## **Granularity**

Granularity refers to the lowest (or most granular) level of information stored in any table. If a table contains sales data for each and every day, then it has a daily granularity. If a table contains total sales data for each month, then it has monthly granularity.

## **Level of Granularity of Fact Table?**

A fact table is usually designed at a low level of Granularity. This means that we need to find the lowest level of information that can store in a fact table. E.g.Employee performance is a very high level of granularity. Employee\_performance\_daily, employee\_perfomance\_weekly can be considered lower levels of granularity.  
The granularity is the lowest level of information stored in the fact table. The depth of data level is known as granularity. In date dimension, the level could be year, month, quarter, period, week, day of granularity.  
The process consists of the following two steps:  
– Determining the dimensions that are to be included  
– Determining the location to locate the hierarchy of each dimension of information. The above factors of determination will be resent to the requirements.

## **Fact Table**

A Fact table is a table that contains **measurements** along the attributes of dimension tables. It can contain the information at lowest possible level. Some fact table just contains summary data, called as **Aggregated Fact Table**. The fact table almost contains the **date stamped** data. Let us discuss the characteristics of a fact table. Facts are typically (but not always) numerical values that can be aggregated.

**Concatenated Key**  
Fact table contains Concatenated key which is the concatenation of primary keys of all the dimension tables. The concatenated key of fact table must uniquely identify the row in a fact table.

**Data Grain**  
Data grain shows how deep the measurements in fact table have been stored. Data grain must be at the possible highest level.

**Additive Measures**  
Attributes of the fact table can be**fully additive** or **semi-additive**. Fully additive measures are those that can be easily summed up for all dimensions in fact table. For example: quantity\_ordered, is an attribute that can be summed up for all dimensions. Like, we can take out total quantity\_order, for a particular customer, region, date, brand, etc.  Semi-additive measures are those which can be summed along some dimensions of fact table but not all dimensions. Like, balance amount cannot be summed up over time dimension as it changes over the time.

**Sparse Data**  
Sometimes we may see the records in fact table that has **attributes with** **null measures**.  For example, there may not be any order on a holiday. So, attributes for this date will have null measures. We do not have to store measure for such kind of records as it does not provide any information.

**Degenerated dimensions**  
Sometimes you may come across some dimensions in fact table, which are not additive at all. For example, order\_number, customer\_id, you cannot add these types of dimensions. However, in case, you need to find order made by a particular customer in this month; then you will need the customer\_id to relate your search. These types if attributes or dimensions of fact table are called **Degenerated Dimension**.

## **Dimension Table**

Dimension Table is a key component for Star Schema. A dimension table contains the attributes which represent dimensions, along which the measurement is taken in fact table. Dimensions are mutually independent. Technically speaking, a dimension is a data element that categorizes each item in a data set into non-overlapping regions. Dimensions are the object or context. That is - dimensions are the 'things' about which something is being spoken.

**Attributes and Keys**  
Every Dimension table must have a **primary key** that uniquely identifies each record of the table. It is commonly observed that the dimension table contain many attributes. Hence, it appears to be **wide** i.e. when you create a dimension table you will find it spreading **horizontally**.

**Attribute values**  
The values of the attributes in dimension table are rarely numeric, most of the times you will find the values in attributes are in **textual format**. For example, product name, brand, category, sub-category, etc.

**Relation among Attributes**  
Frequently you can observe, the attributes you come across in a dimension table are not directly related. Like, Product\_brand has to do nothing with the package\_date but still both could be the attributes of Product dimension table.

**Normalization**  
The dimension table is **not** supposed to be **Normalized**. This is because normalizing a table would create many intermediate tables. When a query picks up an attribute from dimension table and recovers measurements along that for the fact table, the query has to go through those intermediate tables which become inefficient. Hence, dimension tables are not Normalized.

**Drilling down, rolling up**  
Attributes of dimension table allow you to get the details either by traversing from higher level of aggregated attributes to lower level attributes. For example, if you want to find the total sale in a region then you may drill down to find sales by state, city, zip. You can even roll up to find total sales first by the zip, then by city and then state.

**Multiple Hierarchy**  
Often dimension table offers multiple hierarchies. For example, we have a product dimension table for a departmental store. Now, we have two departments marketing and accounting department.

The marketing department will drill down among attributes of product dimension table in a certain hierarchy to obtain measurements for the fact table. On other hands, accounting department will drill down among the attributes of product dimension table in the different hierarchy to obtain measurements for the fact table.

So, dimension table must have multiple hierarchies or level of aggregation of attributes to let user drill down along any of the multiple hierarchies.

**Records**  
Though a dimension table has too many attributes, it has fewer records.

## **Types of Dimensions**

In a data warehouse model, dimension can be of following types,

* Conformed Dimension
* Junk Dimension
* Degenerated Dimension
* Role Playing Dimension

Based on how frequently the data inside a dimension changes, we can further classify dimension as

* Unchanging or static dimension (UCD)
* Slowly changing dimension (SCD)
* Rapidly changing Dimension (RCD)

## **Conformed Dimension**

A conformed dimension is the dimension that is shared across multiple subject areas. Consider the 'Customer' dimension. Both marketing and sales departments may use the same customer dimension table in their reports. Similarly, a 'Time' or 'Date' dimension will be shared by different subject areas. These dimensions are conformed dimensions. Theoretically, two dimensions which are either identical or strict mathematical subsets of one another are said to be conformed. They are used to compare the measures from each star schema. The reuse of conformed dimensions is very common in order to “support true, cross-business process analysis.

A Dimension that is utilized as a part of different areas is called a conformed dimension. It might be utilized with different fact tables in a single database or over numerous data marts/warehouses. For example, if subscriber dimension is connected to two fact tables – billing and claim then the subscriber dimension would be treated as conformed dimension.

A conformed dimension is the dimension that is shared across multiple stars (subject area).

Conformed dimensions are the dimensions which can be used across multiple Data Marts in combination with multiple facts tables accordingly. A conformed dimension is a dimension that has exactly the same meaning and content when being referred from different fact tables. A conformed dimension can refer to multiple tables in multiple data marts within the same organization.

## **Junk Dimension**

A junk dimension is a grouping of typically low-cardinality attributes (flags, indicators etc.) so that those can be removed from other tables and can be junked into an abstract dimension table. These junk dimension attributes might not be related. The only purpose of this table is to store all the combinations of the dimensional attributes which you could not fit into the different dimension tables otherwise. It is a dimension table comprising of attributes that don’t have a place in the fact table or in any of the current dimension tables. Generally, these are the properties like flags or indicators.

A junk dimension is a grouping of typically low-cardinality attributes (flags, indicators etc.) so that those can be removed from other tables and can be junked into an abstract dimension table.

These junk dimension attributes might not be related. The only purpose of this table is to store all the combinations of the dimensional attributes which you could not fit into the different dimension tables otherwise. Junk dimensions are often used to implement [Rapidly Changing Dimensions](https://dwbi.org/data-modelling/dimensional-model/20-implementing-rapidly-changing-dimension.html) in data warehouse.

For example, it can be member eligibility flag set as ‘Y’ or ‘N’ or any other indicator set as true/false, any specific comments, etc. if we keep all such indicator attributes in the fact table then its size gets increased. So, we combine all such attributes and put in a single dimension table called as junk dimension having unique junk IDs with a possible combination of all the indicator values.

Junk dimensions are often used to implement [Rapidly Changing Dimensions](https://dwbi.org/data-modelling/dimensional-model/20-implementing-rapidly-changing-dimension.html) in data warehouse.

* In scenarios where certain data may not be appropriate to store in the schema, this data (or attributes) can be stored in a junk dimension. The nature of data of junk dimension is usually Boolean or flag values.
* A single dimension is formed by lumping a number of small dimensions. This dimension is called a junk dimension. Junk dimension has unrelated attributes. The process of grouping random flags and text attributes in dimension by transmitting them to a distinguished sub dimension is related to junk dimension.

## **Role Playing Dimension**

Dimensions are often reused for multiple applications within the same database with different contextual meaning. For instance, a "Date" dimension can be used for "Date of Sale", as well as "Date of Delivery", or "Date of Hire". This is often referred to as a 'role-playing dimension'.

These are the dimensions which are utilized for multiple purposes in the same database. For example, a date dimension can be used for “Date of Claim”, “Billing date” or “Plan Term date”. So, such a dimension will be called as Role playing dimension. The primary key of Date dimension will be associated with multiple foreign keys in the fact table.

## **Degenerated Dimension**

A degenerated dimension is a dimension that is derived from a fact table and does not have its own dimension table. A dimension key, such as transaction number, receipt number, Invoice number etc. does not have any more associated attributes and hence cannot be designed as a dimension table.

A degenerated dimension is a dimension which is not a fact but present in the fact table as a primary key. It does not have its own dimension table. We can also call it a single attribute dimension table. But, instead of keeping it separately in a dimension table and putting an additional join, we put this attribute in the fact table directly as a key. Since it does not have its own dimension table, it can never act a foreign key in fact table.

## **Mini Dimension?**

Mini dimensions can be used to handle rapidly changing dimension scenarios. If a dimension has a huge number of rapidly changing attributes it is better to separate those attributes in a different table called mini dimension. This is done because if the main dimension table is designed as SCD type 2, the table will soon outgrow in size and create performance issues. It is better to segregate the rapidly changing members in different tables thereby keeping the main dimension table small and performing. When **Mini Dimension starts changing rapidly**, multiple Mini Dimensions can be introduced to handle such scenarios. If **no fact records are to associate main dimension and mini dimension,** a fact less fact table can be used associate main dimension and mini dimension.

Mini dimensions contain the rapidly changing attributes of the original dimension and are treated as a stand-alone dimension. This dimension will band the attributes together instead of treating them as individual pieces of data. The fact table can then connect to the mini-dimension to base upon the data provided or this mini-dimension could be connected to the “parent” dimension using the surrogate key as a Type 1 or Type 2 SCD attribute.

## **Slowly Changing Dimension (SCD)?**

SCD (Slowly changing dimensions), are the dimensions in which the data changes slowly, rather than changing regularly on a time basis. These can be of many types, e.g. Type 0, Type 1, Type 2, Type 3 and Type 6, although Type 1, 2 and 3 are most common. These are most important amongst all the dimensions. These are the dimensions where attribute values vary with time. These can be of many types, e.g. Type 0, Type 1, Type 2, Type 3, Type 4 and Type 6

Three types of SCDs are used in data warehousing, which are defined as:   
– **SCD1**: It is a record that is used to replace the original record even there is only one record existing in the database. The current data will be replaced and the new data will take its place.  
– **SCD2**: It is the new record file that is added to the dimension table. This record exists in the database with the current data and previous data that is stored in the history.  
– **SCD3**: This uses the original data that is modified to the new data. This consists of two records: one record that exists in the database and another record that will replace the old database record with the new information.

* Type 0: Always retains original
* Type 1: Keeps latest data, old data is overwritten
* Type 2: Keeps the history of old data by adding new row
* Type 3: Adds new attribute to store changed value
* Type 4: Use separate history table
* Type 6: combination of type 1, 2 and 3

**Type 0:**

A Type 0 dimension is where dimensional changes are not considered. This does not mean that the attributes of the dimension do not change in actual business situations. It just means that, even if the value of the attributes change, history is not kept and the table holds all the previous data. These are the dimensions where attribute value remains steady with time. **For example,** Subscriber’s DOB is a type-0 SCD because it will always remain the same irrespective of the time.

**Type 1:**

A type 1 dimension is where history is not maintained and the table always shows the recent data. This effectively means that such a dimension table is always updated with recent data whenever there is a change, and because of this update, we lose the previous values. These are the dimensions where the previous value of the attribute is replaced by the current value. **For example,** Subscriber’s address (where the business requires to keep the only current address of subscriber) can be a Type-1 dimension.

**Type 2:**

A type 2 dimension table tracks the historical changes by creating separate rows in the table with different surrogate keys (unlimited history is preserved). Consider there is a customer C1 under group G1 first and later on the customer is changed to group G2. Then there will be two separate records in the dimension table. Note that separate surrogate keys are generated for the two records. NULL end date in the second row denotes that the record is the current record. Also note that, instead of start and end dates, one could also keep version number column (1, 2 … etc.) to denote different versions of the record. For **example,** Subscriber’s address (where the business requires to keep a record of all the previous addresses of the subscriber). In this case, multiple rows for a subscriber will be inserted in the table with his/her different addresses. There will be some column(s) that will identify the current address. **For example,** ‘start date’ and ‘End date’. The row where ‘End date’ value will be blank would contain the subscriber's current address and all other rows will be having previous addresses of the subscriber.

**Type 3:**

A type 3 dimension stored the history in a separate column instead of separate rows. So unlike a type 2 dimension which is vertically growing, a type 3 dimension is horizontally growing. This is only good when you need not store many consecutive histories and when date of change is not required to be stored. These are the types of dimensions where limited history is preserved. And we use an additional column to maintain the history. **For example,** Subscriber’s address (where the business requires to keep a record of current & just one previous address). In this case, we can dissolve the ‘address’ column into two different columns – ‘current address’ and ‘previous address’. So, instead of having multiple rows, we will be having just one row showing current as well as the previous address of the subscriber.

**Type-4:**

In this type of dimension, the historical data is preserved in a separate table. The main dimension table holds only the current data. **For example,** the main dimension table will have only one row per subscriber holding its current address. All other previous addresses of the subscriber will be kept in the separate history table. This type of dimension is hardly ever used.

**Type 6:**

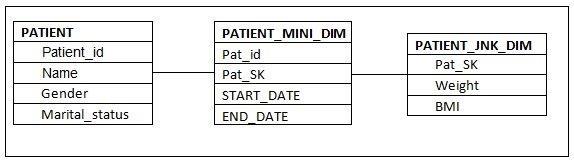
A type 6 dimension is a hybrid of type 1, 2 and 3 (1+2+3) which acts very similar to type 2, but only you add one extra column to denote which record is the current record.

## **Rapidly Changing Dimension (RCD)?**

This is a dimension where data changes rapidly. Rapidly changing dimensions are dimensions where the attribute values of the dimension change frequently causing the dimension to grow rapidly if you have designed the dimension to capture the changes as a Type 2 dimension. A dimension is a fast changing or rapidly changing dimension if one or more of its attributes in the table changes very fast and in many rows. The rapid growth of this dimension will impact maintenance and performance as the dimension grows. Handling rapidly changing dimensions in a data warehouse is very difficult because of many performance implications.

As you know, slowly changing dimension type 2 is used to preserve the history for the changes. But the problem with type 2 is, with each and every change in the dimension attribute, it adds a new row to the table. If in case there are dimensions that are changing a lot, tables become larger and may cause serious performance issues. Hence, use of the type 2 may not be the wise decision to implement the rapidly changing dimensions.

Consider the fact table, in which not all the attributes of the table changes rapidly. There may be some attributes that may be changing rapidly and others not. The idea here is to **separate the rapidly changing attribute from the slowly changing ones and move those attributes to another table called junk dimension** and maintain the slowly changing attribute in the same table. In this way, we can handle situations of increasing table size. In this step, we must link the junk dimension and patient table. Keep in mind; we cannot simply refer to the junk dimension table by adding its primary key to the patient table as foreign key. Because any changes made to the junk dimension will have to reflect in the patient table, this obviously increases the data in patient dimension. Instead, we create one more table called mini dimension that acts as a bridge between Patient and Junk dimension. We can also add the columns such as start and end date to track the change history. Below is the structure of the mini dimension:



## **Degenerated dimension**

A degenerated dimension is a dimension that is derived from a fact table and does not have its own dimension table. A dimension key, such as transaction number, receipt number, Invoice number etc. does not have any more associated attributes and hence cannot be designed as a dimension table.

## **Fact-less fact**

A fact table that does not contain any measure is called a fact-less fact. This table will only contain keys from different dimension tables. This is often used to resolve a many-to-many cardinality issue. These tables are used to capture the action of the business process. For example, a criminal case is a simple fact with no measures but can have a lot of dimensional attributes associated with the fact.

**For example,** suppose you are maintaining an employee attendance record system, you can have a factless fact table having three keys – Employee\_ID, Department\_ID, Time\_ID. You can see that the above table does not contain any measure. Now if you want to answer the question below, you can do it easily using the above single factless fact table rather than having two separate fact tables: *“How many employees of a particular department were present on a particular day?”* So, a factless fact table offers flexibility to the design.

**Explanatory Note:**

Consider a school, where a single student may be taught by many teachers and a single teacher may have many students. To model this situation in a dimensional model, one might introduce a factless-fact table joining teacher and student keys. Such a fact table will then be able to answer queries like,

1. Who are the students taught by a specific teacher?
2. Which teacher teaches maximum students?
3. Which student has the highest number of teachers, etc. etc.

## **Coverage fact**

A fact-less-fact table can only answer 'optimistic' queries (positive query) but cannot answer a negative query. Again, consider the illustration in the above example. A fact-less fact containing the keys of tutors and students cannot answer a query like below,

1. Which teacher did not teach any student?
2. Which student was not taught by any teacher?

Why not? Because factless fact table only stores the positive scenarios (like student being taught by a tutor) but if there is a student who is not being taught by a teacher, then that student's key does not appear in this table, thereby reducing the coverage of the table.

Coverage fact table attempts to answer this - often by adding an extra flag column. Flag = 0 indicates a negative condition and flag = 1 indicates a positive condition. To understand this better, let's consider a class where there are 100 students and 5 teachers. So, coverage fact table will ideally store 100 X 5 = 500 records (all combinations) and if a certain teacher is not teaching a certain student, the corresponding flag for that record will be 0.

## **Snapshot Fact**

A snapshot fact always shows the latest (or last known) state of the measures. The latest records of a SCD type 2 dimension also do the same. The only difference is that a fact shows the state of the measures whereas a SCD Type 2 table shows the state of the attributes.

## **Star-schema**

* This schema is used in data warehouse models where one centralized fact table references a number of dimension tables so as the keys (primary key) from all the dimension tables flow into the fact table (as foreign key) where measures are stored. This entity-relationship diagram looks like a star, hence the name.
* Star schema is the most simple kind of schema where one fact table is present in the center and all the dimension tables are connected only with the fact table and no dimension table is connected with any other dimension table.
* In a Star schema design, any information can be obtained just by traversing a single join, which means this type of schema will be ideal for information retrieval (faster query processing). Here, note that all the hierarchies (or levels) of the members of a dimension are stored in the single dimension table.
* Star schema is probably the most popular schema in dimensional modeling because of its simplicity and flexibility.
* Star schemas are better for [data marts](https://www.xplenty.com/glossary/what-is-data-mart/) with simple relationships.

## **Snowflake Schema**

* Snowflake schema is just like star schema but the difference is, here one or more dimension tables are connected with other dimension tables as well as with the central fact table.
* Dimensions with hierarchies can be decomposed into a snowflake structure when you want to avoid joins to big dimension tables when you are using an aggregate of the fact table. A centralized fact table references a number of other dimension tables; however, those dimension tables are further normalized into multiple related tables.
* Snow flaking increases the level of normalization in the data. This has obvious disadvantages in terms of information retrieval since we need to read more tables (and traverse more SQL joins) to get the same information.
* There are two main advantages to the snowflake schema: Better data quality (data is more structured, so data integrity problems are reduced) Less disk space is used then in a denormalized model.
* Snowflake schemas have no redundant data, so they're easier to maintain.
* Snowflake schemas are good for [data warehouses](https://www.xplenty.com/glossary/what-is-data-warehouse/).

## **Keys**

* **Primary key:** The Primary key is an attribute in a table that can uniquely identify each record in a table. It is compulsory for every table. One of the candidate keys is selected as most important and becomes the primary key. The primary key creates the cluster index automatically, but the Unique key does not.
* **Candidate key:** The Candidate key is an attribute or set of an attribute which can uniquely identify a tuple. The Primary key can be selected from these attributes. A [Candidate key](http://en.wikipedia.org/wiki/Candidate_key) is minimal superkey, i.e., no proper subset of Candidate key attributes can be a superkey.
* **Super key:** Super key is a superset of the candidate key which can uniquely identify a tuple. A [superkey](http://en.wikipedia.org/wiki/Superkey)is a set of attributes of a relation schema upon which all attributes of the schema are functionally dependent. No two rows can have the same value of super key attributes.
* **Foreign key:** The Foreign key is a primary key from one table, which has a relationship with another table. It acts as a cross-reference between tables. [Foreign key](http://en.wikipedia.org/wiki/Foreign_key) is a field (or collection of fields) in one table that uniquely identifies a row of another table.
* **Surrogate key:** A surrogate key is any column or set of columns that can be declared as the primary key instead of a "real" or natural key. Sometimes there can be several natural keys that could be declared as the primary key, and these are all called candidate keys. So, a surrogate is a candidate key. The surrogate key is generated when a new record is inserted into a table. When a primary key is generated at runtime, it is called a surrogate key. A surrogate key is typically a numeric value. They are used to relate the facts in the fact table to the appropriate rows in the dimension tables and are used to connect the fact to the dimension tables of a data warehouse.
* **Natural key**: A natural key is a column value that has a relationship with the rest of the column values in a given data record. Here are some examples of natural keys values: Social Security Number, ISBN, and TaxID. Sometimes the primary key is made up of real data and these are normally referred to as natural keys. They are tied to the business setting and requirements and if these would change (e.g., due to a merger or acquisition, a new legislation) then all tables using those keys need to be updated, which may be a resource intensive operation in a data warehouse environment, because not only the current state is stored, but also historical data.
* **Prime key:** Attribute that forms candidate keys. The attributes of the candidate key which defines the uniqueness (Eg: SSN number in an employee database). Attributes of the relation which exist in at least one of the possible candidate keys, are called prime or key attributes.
* **Non - Prime key:** Attributes of the relation which do not exist in any of the possible candidate keys of the relation, such attributes are called non-prime or non-key attributes. In order to distinguish whether an attribute of the relation is prime or non-prime, we should always find all the possible candidate keys of the given relation.

## **Why do we need to replace primary key with surrogate key?**

A natural key and a surrogate key are two types of primary key. A natural key is a single column or a combination of columns that has a business value and occurs naturally in the real world (e.g. Social security number, International Standard Book Number…). A surrogate key in SQL Server is created by assigning an identity property to a column that has a number data type. A surrogate key is a value generated right before the record is inserted into a table.

There are several reasons to replace a natural key with a surrogate key. A surrogate key can be used to reduce coupling since it doesn’t have a business value and it’s not coupled with any external application connected to a database. For example, if the business logic changes at some point, that would require updating a natural key across all foreign key relationships. In some cases, even using Social Security Number or International Standard Book Number as a natural key doesn’t guarantee that a primary key will be unique. A surrogate key value is unique and since it doesn’t have a business value it won’t be updated over time.

Also, using a surrogate key may increase performance because a large natural key can degrade database performance, and due to a fact that surrogate keys are usually integer values a smaller index on a primary key will have better performance on JOIN operations. In case of combined natural primary keys JOIN operations can became very complex:

CREATE TABLE AS does not inherits IDENTITY property from parent table. In that case creating a VIEW over the table using the same ROW\_NUMBER window function would be the perfect choice.

## **Data Independence**

Data independence specifies that "the application is independent of the storage structure and access strategy of data." It makes you able to modify the schema definition at one level without altering the schema definition in the next higher level.

**There are two types of Data Independence**:

**Physical Data Independence:** Physical data is the data stored in the database. It is in the bit-format. Modification at the physical level should not affect the logical level.

For example: If we want to manipulate the data inside any table that should not change the format of the table.

**Logical Data Independence:** Logical data in the data about the database. It basically defines the structure. Such as tables stored in the database. Modification at the logical level should not affect the view level.

For example: If we need to modify the format of any table, that modification should not affect the data inside it.

## **Integrity rules in DBMS**

Data integrity is one significant aspect while maintaining the database. So, data integrity is enforced in the database system by imposing a series of rules. Those set of integrity is known as the integrity rules.

**There are two integrity rules in DBMS:**

**Entity Integrity**: It specifies that "Primary key cannot have a NULL value."

**Referential Integrity**: It specifies that "Foreign Key can be either a NULL value or should be the Primary Key value of other relation

## **Functional Dependency**

Functional Dependency is the starting point of normalization. It exists when a relation between two attributes allow you to determine the corresponding attribute's value uniquely. The functional dependency is also known as database dependency and defines as the relationship which occurs when one attribute in a relation uniquely determines another attribute. It is written as A->B which means B is functionally dependent on A. *We need primary key to fetch data from a Student table and every other column****depends****on it or can be fetched using it.*

## **Partial Dependency**

Now as we just discussed that the primary key for this table is a composition of two columns which is student\_id & subject\_id but the teacher's name only depends on subject, hence the subject\_id, and has nothing to do with student\_id.

This is **Partial Dependency**, where an attribute in a table depends on only a part of the primary key and not on the whole key. To remove Partial dependency, we can divide the table, remove the attribute which is causing partial dependency, and move it to some other table where it fits in well.

## **Transitive Dependency**

When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key. This is **Transitive Dependency**.

What about our second new column total\_marks? Does it depend on our Score table's primary key?

Well, the column total\_marks depends on exam\_name as with exam type the total score changes. For example, practicals are of less marks while theory exams are of more marks.

But, exam\_name is just another column in the score table. It is not a primary key or even a part of the primary key, and total\_marks depends on it.

The advantage of removing transitive dependency is:

* Amount of data duplication is reduced.
* Data integrity achieved.

## **Multi-valued Dependency**

A table is said to have multi-valued dependency, if the following conditions are true,

* For a dependency A → B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency.
* Also, a table should have at-least 3 columns for it to have a multi-valued dependency.
* And, for a relation R(A,B,C), if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.

If all these conditions are true for any relation(table), it is said to have multi-valued dependency.

## **Normalization**

Database normalization is the process of structuring a [database](https://en.wikipedia.org/wiki/Database), usually a [relational database](https://en.wikipedia.org/wiki/Relational_database), in accordance with a series of so-called [normal forms](https://en.wikipedia.org/wiki/Database_normalization#Normal_forms) in order to reduce [data redundancy](https://en.wikipedia.org/wiki/Data_redundancy) and improve [data integrity](https://en.wikipedia.org/wiki/Data_integrity).

Normalization entails organizing the [columns](https://en.wikipedia.org/wiki/Column_(database)) (attributes) and [tables](https://en.wikipedia.org/wiki/Relation_(database)) (relations) of a database to ensure that their [dependencies](https://en.wikipedia.org/wiki/Dependency_theory_(database_theory)) are properly enforced by database integrity constraints. It is accomplished by applying some formal rules either by a process of synthesis (creating a new database design) or decomposition (improving an existing database design).

Normalization is a database design technique that **reduces data redundancy** and **eliminates** undesirable characteristics **like Insertion, Update and Deletion Anomalies**. Normalization rules divide larger tables into smaller tables and link them using relationships. The purpose of Normalization in SQL is to eliminate redundant (repetitive) data and ensure data is stored logically.

It was first proposed by [Edgar F. Codd](https://en.wikipedia.org/wiki/Edgar_F._Codd) as part of his [relational model](https://en.wikipedia.org/wiki/Relational_model). The inventor of the [relational model](https://www.guru99.com/relational-data-model-dbms.html) Edgar Codd proposed the theory of normalization of data with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.

Insertion Anomaly

Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as NULL. Also, if we have to insert data of 100 students of the same branch, then the branch information will be repeated for all those 100 students. These scenarios are nothing but Insertion anomalies.

Updation Anomaly

What if Mr. X leaves the college? or is no longer the HOD of the computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

Deletion Anomaly

In our Student table, two different information are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is Deletion anomaly.

### **1NF?**

**1NF** is the **First Normal Form**. It is the simplest type of normalization that you can implement in a database. The primary objectives of 1NF are:

* Every column must have atomic (single value).
* To Remove duplicate columns from the same table.
* Create separate tables for each group of related data and identify each row with a unique column.
* Values stored in a column should be of the same domain.
* All the columns in a table should have unique names.
* And the order in which data is stored, does not matter.

### **2NF?**

**2NF** is the **Second Normal Form**. A table is said to be 2NF if it follows the following conditions:

* The table is in 1NF, i.e., firstly it is necessary that the table should follow the rules of 1NF.
* Every non-prime attribute is fully functionally dependent on the primary key, i.e., every non-key attribute should be dependent on the primary key in such a way that if any key element is deleted, then even the non-key element will still be saved in the database.
* And, it should not have Partial Dependency.

### **3NF?**

**3NF** stands for **Third Normal Form**. A database is called in 3NF if it satisfies the following conditions:

* It is in second normal form.
* There is no transitive functional dependency.
* For example: X->Z

**Where:**  
X->Y  
Y does not -> X  
Y->Z so, X->Z

### **BCNF?**

**BCNF** stands for **Boyce-Codd Normal Form**. It is an advanced version of 3NF, so it is also referred to as 3.5NF. BCNF is stricter than 3NF. This form deals with a certain type of anomaly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF.

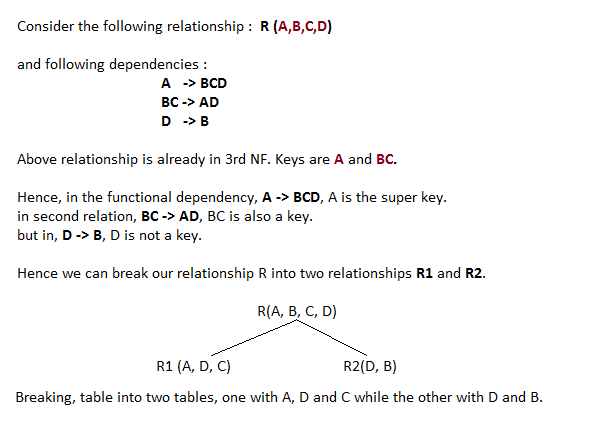
A table complies with BCNF if it satisfies the following conditions:

* It is in 3NF.
* For every functional dependency X->Y, X should be the super key of the table. It merely means that X cannot be a non-prime attribute if Y is a prime attribute.

In the table above, student\_id, subject form primary key, which means subject column is a **prime attribute**.

But, there is one more dependency, professor → subject.

And while subject is a prime attribute, professor is a **non-prime attribute**, which is not allowed by BCNF. To make this relation(table) satisfy BCNF, we will decompose this table into two tables, **student** table and **professor** table.



### **4NF?**

Fourth Normal Form comes into picture when **Multi-valued Dependency** occurs in any relation. A table is said to be in the Fourth Normal Form when,

* It is in the Boyce-Codd Normal Form.
* And, it doesn't have Multi-Valued Dependency.

A table can also have functional dependency along with multi-valued dependency. In that case, the functionally dependent columns are moved in a separate table and the multi-valued dependent columns are moved to separate tables.

If no database table instance contains two or more, independent and multivalued data describing the relevant entity, then it is in 4th Normal Form.

### **5NF?**

A table is in 5th Normal Form only if it is in 4NF and it cannot be decomposed into any number of smaller tables without loss of data.

### **6NF?**

6th Normal Form is not standardized, yet however, it is being discussed by database experts for some time. Hopefully, we would have a clear & standardized definition for 6th Normal Form in the near future.

## **Denormalization**

Denormalization is the process of boosting up database performance and adding of redundant data which helps to get rid of complex data. Denormalization is a part of database optimization technique. This process is used to avoid the use of complex and costly joins. The un-normalized and de-Normalized database are completely different from each other. Before the process of de-normalization of any database, that should be normalized firstly. In this process, firstly the redundancy of the data will be removed using normalization process than through denormalization process we will add redundant data as per the requirement so that we can easily avoid the costly joins. The de-Normalization process enhances the read performance while some degradation occurs in write performance. It can be achieved by making a group of data in the redundant form. It covers the inefficiencies in the relational database software.

Denormalization can be successfully performed only with thorough knowledge of the application and should be performed only if performance issues indicate that it is needed. One of the things to consider when you denormalize is the amount of effort it will then take to keep your data up-to-date with changes. If your purpose is an easy maintenance of data, less redundancy, and efficient access then you should go with a de-normalized database.

**When and why to use denormalization:**

1. Maintaining history
2. Improving query performance
3. Speeding up reporting
4. Computing commonly needed values up front

**Examples include the following:**

* Materialized views for implementation purpose
* Storing the count of “many” objects in a one-to-many relationship.
* Linking attribute of one relation with other relations.
* To improve the performance and scalability of web applications.

**Pros of Denormalization:**

1. Minimizing the need for joins: Retrieving data is faster since we do fewer joins.
2. Reducing the number of tables, in some cases: Queries to retrieve can be simpler(and therefore less likely to have bugs), since we need to look at fewer tables.
3. Precomputing aggregate values, that is, computing them at data modification time, rather than at select time.

**Cons of Denormalization:**

1. Updates and inserts are more expensive. Denormalization usually speeds retrieval but can slow updates and can make update and insert code harder to write.
2. Possibility of data anomaly: The amount of data needed to get modified increased during the process of denormalization.
3. Data may be inconsistent. Which is the “correct” value for a piece of data?
4. Data redundancy necessities more storage. Denormalization can increase the size of tables.
5. Denormalization is always application-specific and needs to be re-evaluated if the application changes.

## **Denormalization Techniques?**

There are plenty of techniques through which the database can be denormalized. We have explained about each of them in brief below:

* ***Storing derivable information***

This technique is perfect to follow if you want to execute calculations repeatedly while queries. The derived calculation needs to be stored in the master table if there are detailed records in it. They are perfect to use if you require derivable values more frequently or you don’t prefer to alter source values regularly.

* ***Moving data fields into a supertype***

In this type of technique, the data fields of all the subtypes are moved into supertypes. The new model looks completely different from the normalized model. The subtypes are completely removed from the data fields using this technique.

* ***Using hardcore values***

This technique is perfect to use if there are reference table along with consistent records. There will be no need to join the tables for getting the reference values. You need to check out the constraint for validating values with this technique. You have to rewrite a new value each time in the table.

* ***Copying data field from an entity***

In this technique, there is no removal of the entity performed by the user. Instead, the data fields copied from one entity to another. This technique consumes very less time and boosts the retrieval speed.

***Things to keep in mind during the denormalization process***

Denormalization is a very complicated process that demands pretty serious effort and expertise. That’s why it is important that you should follow the below mentioned tips for performing this task adequately.

* Always target those components in the database that you want to boost. You should avoid trying denormalizing the complete database.
* You should have adequate learning about the logical design of the application to have a complete idea about the components affected greatly by the denormalization.
* Knowing the data storage techniques will be a plus point. It will help in picking you the most relevant one.
* The modification of data in the application needs to be analyzed carefully. The reason behind it is that the maintenance of the database’s integrity becomes difficult if data changes too frequently.

## **DDL, DML, DCL**

DDL stands for Data Definition Language. SQL queries like CREATE, ALTER, DROP and RENAME come under this.  
DML stands for Data Manipulation Language. SQL queries like SELECT, INSERT and UPDATE come under this.  
DCL stands for Data Control Language. SQL queries like GRANT and REVOKE come under this. DCL languages are used to control the user access to the database, tables, views, procedures, functions and packages. They give different levels of access to the objects in the database.

## **Joins in SQL**

An SQL Join is used to combine/retrieving data from two or more tables, based on a common field between them. The tables are mutually related using primary and foreign keys. Basically, we have only three types of joins: Inner join, Outer join and Cross join. Different types of Joins are:

* **INNER JOIN**: An inner join using either of the equivalent queries gives the intersection of the two tables, i.e. the two rows they have in common. The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies i.e value of the common field will be same. Inner join returns only those records/rows that match/exists in both the tables.
* **LEFT JOIN**: - A left outer join will give all rows in A, plus any common rows in B. This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.
* **RIGHT JOIN**: RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.
* **FULL JOIN**: A full outer join will give you the union of A and B, i.e. All the rows in A and all the rows in B. If something in A doesn't have a corresponding datum in B, then the B portion is null, and vice versa. FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. Initially, it applies **inner join** on **Table\_A** and **Table\_B** to retrieve matching tuples from both the tables. Then it extends those tuples of **Table\_A** with **NULL** that do not have a matching tuple in Table\_B. Further, it extends those tuples from **Table\_B** with **NULL** that do not have a matching tuple in Table\_A.
* **CARTESIAN/CROSS JOIN**: The CARTESIAN JOIN is also known as CROSS JOIN. Cross join is a cartesian join means cartesian product of both the tables. This join does not need any condition to join two tables. This join returns records/rows that are multiplication of record number from both the tables means each row on left table will related to each row of right table. In a CARTESIAN JOIN there is a join for each row of one table to every row of another table. This usually happens when the matching column or WHERE condition is not specified.
* In the absence of a WHERE condition the CARTESIAN JOIN will behave like a CARTESIAN PRODUCT i.e., the number of rows in the result-set is the product of the number of rows of the two tables.
* In the presence of WHERE condition this JOIN will function like an INNER JOIN.
* Generally speaking, Cross join is similar to an inner join where the join-condition will always evaluate to True
* **SELF JOIN**: In SELF JOIN a table is joined to itself. That is, each row of the table is joined with itself and all other rows depending on some conditions. Self join is used to join a database table to itself, particularly when the table has a Foreign key that references its own Primary Key.
* **NATURAL JOIN**: Natural join is a type of equi join which occurs implicitly by comparing all the same names columns in both tables. The join result have only one column for each pair of equally named columns. The associated tables should have one or more pairs of identically named columns of same data type. Don’t use ON clause in a natural join.
* **EQUI JOIN**: EQUI JOIN performs a JOIN against equality or matching column(s) values of the associated tables and an equal sign (=) is used as comparison operator in the where clause to refer equality. You may also perform EQUI JOIN by using JOIN keyword followed by ON keyword and then specifying names of the columns along with their associated tables to check equality. SQL EQUI JOIN is a specific type comparison base join (equally comparison) not allowing other comparison operator such as <, > <= etc. And create record set result that are combining columns value from the tables (two or more table).
* **NON-EQUI JOIN**: It is a sql join condition which makes use of some comparison operator other than the equal sign like >, <, >=, <=.

## **Anti Joins in Oracle**

Anti-join is used to make the queries run faster. ... Anti-join between two tables returns rows from the first table where no matches are found in the second table. It is opposite of a semi-join. An anti-join returns one copy of each row in the first table for which no match is found.

* **LEFT ANTI JOIN:** A *left anti join*, which brings in only rows from the left table that don't have any matching rows from the right table.
* **RIGHT ANTI JOIN:** A *right anti join*, which brings in only rows from the right table that don't have any matching rows from the left table.
* **SEMI JOIN:** Semi-join is introduced in Oracle 8.0. It provides an efficient method of performing a WHERE EXISTS sub-query. A semi-join returns one copy of each row in first table for which at least one match is found. Semi-joins are written using the EXISTS construct.

## **View**

A [view](http://en.wikipedia.org/wiki/View_(SQL))is a virtual table based on the result-set of an SQL statement. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition. We can create using create view syntax.

CREATE VIEW view\_name AS

SELECT column\_name(s)

FROM table\_name

WHERE condition

Views are a logical virtual table created by [“select query”](http://javarevisited.blogspot.sg/2011/10/selct-command-sql-query-example.html) but the result is not stored anywhere in the disk and every time we need to fire the query when we need data, so always we get updated or latest data from original tables.

Performance of the view depends on our select query. If we want to improve the performance of view we should avoid using join statement in our query or if we need multiple joins between table always try to use the index based column for joining as we know index based columns are faster than a non-index based column. View also allows storing the definition of the query in the database itself.

View is a **virtual table**, created using **Create View** command. This virtual table contains the data retrieved from a **query expression**, in Create View command. View can be created from one or more than one base tables or views. A view can be queried like you query the original base tables.

A view is **computed** each time it is used or accessed. Whenever a view is used the query expression in Create View command is executed at that particular moment. Hence, you always get the **updated** data in a View.

If you update any content in View, it is reflected in the original table, and if any changes had been done to the original base table, it would reflect in its View. But this makes the performance of a View **slower**. For example, a view is created from the join of two or more tables. In that case, you have to pay time to resolve Joins each time a View is used.

But it has some **advantages** like it do **not** require **storage space**. You can create a **customized** view of a complex database. You can **restrict** the user from accessing sensitive information in a database. Reduces the **complexity** of queries by getting data from several tables into a single customized View.

[Views are a great way to hide the inner workings of a query](https://www.essentialsql.com/what-is-a-relational-database-view/) and allow you to just focus on the results.  This is especially true when you’re working with data involving multiple joins.

## **Uses of View**

1. Views can represent a subset of the data contained in a table; consequently, a view can limit the degree of exposure of the underlying tables to the outer world: a given user may have permission to query the view, while denied access to the rest of the base table.
2. Views can join and simplify multiple tables into a single virtual table.
3. Views can act as aggregated tables, where the database engine aggregates data (sum, average etc.) and presents the calculated results as part of the data.
4. Views can hide the complexity of data; for example a view could appear as Sales2000 or Sales2001, transparently partitioning the actual underlying table.
5. Views take very little space to store; the database contains only the definition of a view, not a copy of all the data which it presents.
6. Depending on the SQL engine used, views can provide extra security.

## **When can a view be Updated?**

The SQL UPDATE VIEW command can be used to modify the data of a view. All views are not updatable. So, UPDATE command is not applicable to all views. An updatable view is one which allows performing an UPDATE command on itself without affecting any other table. Like a View created using **DISTINCT** clause, **Group By** clause, **CHECK** constraint (if check constraints violate),**Read-only** option can’t be updated.

1. The view is defined based on one and only one table.
2. The view must include the PRIMARY KEY of the table based upon which the view has been created.
3. The view should not have any field made out of aggregate functions.
4. The view must not have any DISTINCT clause in its definition.
5. The view must not have any GROUP BY or HAVING clause in its definition.
6. The view must not have any SUBQUERIES in its definitions.
7. If the view you want to update is based upon another view, the later should be updatable.
8. Any of the selected output fields (of the view) must not use constants, strings or value expressions.
9. If the view contains joins between multiple tables, you can only insert and update one table in the view, and you can't delete rows.
10. You can't directly modify data in views based on union queries. You can't modify data in views that use GROUP BY or DISTINCT statements.
11. All columns being modified are subject to the same restrictions as if the statements were being executed directly against the base table.
12. Text and image columns can't be modified through views.
13. There is no checking of view criteria. For example, if the view selects all customers who live in Paris, and data is modified to either add or edit a row that does not have City = 'Paris', the data will be modified in the base table but not shown in the view, unless WITH CHECK OPTION is used when defining the view.

**Syntax:**

UPDATE < view\_name > SET<column1>=<value1>,<column2>=<value2>,.....

WHERE <condition>;

#### **Using WITH CHECK OPTION**

The WITH CHECK OPTION clause forces all data-modification statements executed against the view to adhere to the criteria set within the WHERE clause of the SELECT statement defining the view. Rows cannot be modified in a way that causes them to vanish from the view. Listing 9.12 creates a view showing customers from Paris using the WITH CHECK OPTION statement.

CREATE VIEW vwCustomersParis

AS

SELECT CompanyName, ContactName, Phone, City

FROM Customers

WHERE City = 'Paris'

WITH CHECK OPTION

The following Transact-SQL statement attempting to update data by moving everyone from Paris to Lyons will fail because Lyons does not meet the criteria defined in the view. If you did not have WITH CHECK OPTION defined, the UPDATE statement would succeed, and a requery of the view would return no rows. Here's the statement:

UPDATE vwCustomersParis

SET City = 'Lyons'

You may have noticed in Access that placing criteria on the RecordSource query of a form limits the records that are displayed in the form when you open it, but it doesn't limit what records can be added in the form. Using an Access project or an Access database with a form bound to a view that contains WITH CHECK OPTION would allow you to automatically have the criteria enforced for new or updated records.

#### **Updating Views with Joins**

A view that contains a join will only be updateable on one side of the join, unless an INSTEAD OF trigger is created on the view. INSTEAD OF triggers are discussed in the next section. For example, the view shown in Listing 9.13 is based on a join between the Categories and Products tables.

CREATE VIEW vwCategoriesProducts

AS

SELECT Categories.CategoryName,

Products.ProductID, Products.ProductName

FROM Products INNER JOIN

Categories ON

Products.CategoryID = Categories.CategoryID

A view selecting data from both the Categories and Products tables.

The following UPDATE statement will work because it's only affecting the Products table's side of the join:

UPDATE vwCategoriesProducts

SET ProductName = 'Chay'

WHERE ProductID = 1

This UPDATE statement will also work because only affects the Categories table's side of the join:

UPDATE vwCategoriesProducts

SET CategoryName = 'Drinks'

WHERE ProductID = 1

However, the following UPDATE statement attempting to modify columns in both the Products and Categories tables won't work (you'll get the error "View or function 'vwCategoriesProducts' is not updateable because the FROM clause names multiple tables"):

UPDATE vwCategoriesProducts

SET ProductName = 'Chay', CategoryName = 'Drinks'

WHERE ProductID = 1

#### **Using INSTEAD OF Trigger to Update views**

Normally a UNION query is not updateable. However, an INSTEAD OF trigger lets you update the tables involved because it can execute code instead of the default action (UPDATE). The trigger makes use of the inserted table, which contains the new value, to insert data into the appropriate table based on the Type value. It also makes use of the deleted table, which contains the old value, to find the correct record in the base table.

## **Materialized view**

Materialized views are also the logical view of our data-driven by the select query but the result of the query will get stored in the table or disk, also the definition of the query will also store in the database. Materialized View is the **Physical copy** of the original base tables. The Materialized View is like a **snapshot** or **picture** of the original base tables. Like View, it also contains the data retrieved from the **query expression** of **Create Materialized View** command.

When we see the performance of Materialized view it is better than normal View because the data of materialized view will be stored in table and table may be [indexed](http://java67.blogspot.sg/2012/10/difference-between-clustered-vs-nonclustered-index-sql-database.html) so faster for joining, also joining is done at the time of materialized views refresh time so no need to every time fire join statement as in case of view.

But unlike View, the Materialized View are **precomputed** and **stored** on a disk like an object, and they are **not updated** each time they are used. Instead, the materialized view has to be updated **manually** or with the help of **triggers**. The process of updating the Materialized View is called **Materialized View Maintenance**.

## **Indexes**

SQL indexes are the medium of reducing the cost of the query as the high cost of the query will lead to the fall in the performance of the query. An index is used to increase the performance and allow faster retrieval of records from the table. Indexing reduces the number of data pages we need to visit to find a particular data page. Indexing also has a unique value that means that the index cannot be duplicated. Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed. An index or database index is a data structure which is used to quickly locate and access the data in a database table. An index is used to speed up data search and SQL query performance.

An index creates an entry for each value, and it will be faster to retrieve data. For example, suppose you have a book which carries the details of the countries, and you want to find out the information about India than why you will go through every page of that book you could directly go to the index, and then from index you can go to that particular page where all the information about India is given.

A [database index](http://en.wikipedia.org/wiki/Database_index) is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and the use of more storage space to maintain the extra copy of data. Data can be stored only in one order on disk. To support faster access according to different values, faster search like binary search for different values is desired. For this purpose, indexes are created on tables. These indexes need extra space on disk, but they allow faster search according to different frequently searched values.

**Which column to use to build an index?**

Both clustered and non-clustered indexes can be built from one or more table columns. When you create a new table with a primary key in a SQL Server database, a unique clustered index is automatically created on the primary key column. Although this default action is acceptable in most cases, this might not be the optimal clustered index.

The column used for a clustered index should be a unique, identity, or primary key, or any other column where the value is increased for each new entry. As clustered indexes sort the records based on the value, using a column already ordered ascending, such as an identity column, is a good solution.

If a column where new values are not higher than previous is used for a clustered index, adding each new row would require re-ordering, i.e. moving the whole row and placing it to its proper location in accordance with clustered index ordering, thus splitting data pages and affecting SQL Server performance. If such clustered index is created on a table with frequent inserts and updates, it can cause performance degradation.

It’s not recommended to use the primary key as a clustered key without checking whether that is the optimal solution in your scenario first. Also, note the difference between a primary key and clustered index – a primary key can’t have duplicate or null values, while a clustered index can.

Using a unique column for a clustered index enables more efficient search for a specific value. On the other hand, a column that frequently changes its value should not be used for a clustered index. Each change of the column used for the clustered index requires the records to be reordered. This re-ordering can easily be avoided by using a column that is not updated frequently, or not updated at all.

Using a column that stores large data, such as BLOB columns (text, nvarchar(max), image, etc.), and GUID columns is not recommended. Using large values to sort the data is not efficient, and in case of GUID and image columns doesn’t seem to make sense.

A clustered index should not be built on a column already used in a unique index.

Refer this: <https://www.sqlshack.com/top-10-questions-answers-sql-server-indexes/>

Indexes are created using some 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 that the data may or may not be stored in sorted order).
* The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.

There are two main types of indexes in SQL server: Clustered and non-clustered indexes. The clustered index controls the sort of the data pages in the disk, including all the columns in the table, although the index is created by one column only. The non-clustered index does not specify the real data order.

SQL Server Indexes can be categorized also to other types, such as the composite index; which is an index that contains more than one column. Unique index; which ensures the uniqueness of each value in the indexed column or columns as a whole. The last type is the covering index which contains all columns needed for a specific query.

There are two kinds of indices:

1. **Ordered indices:** Indices are based on a sorted ordering of the values.
2. **Hash indices:** Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by function called a hash function.

There is no comparison between both the techniques, it depends on the database application on which it is being applied.

* **Access Types**: e.g. value based search, range access, etc.
* **Access Time**: Time to find particular data element or set of elements.
* **Insertion Time**: 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**: Additional space required by the index.

Indexes have the following properties:

* **Usability**

Indexes are usable (default) or unusable. An **unusable index** is not maintained by DML operations and is ignored by the [**optimizer**](https://docs.oracle.com/cd/E11882_01/server.112/e40540/glossary.htm#CHDGHIIF). An unusable index can improve the performance of bulk loads. Instead of dropping an index and later re-creating it, you can make the index unusable and then rebuild it. Unusable indexes and index partitions do not consume space. When you make a usable index unusable, the database drops its index [**segment**](https://docs.oracle.com/cd/E11882_01/server.112/e40540/glossary.htm#i432714).

* **Visibility**

Indexes are visible (default) or invisible. An **invisible index** is maintained by DML operations and is not used by default by the optimizer. Making an index invisible is an alternative to making it unusable or dropping it. Invisible indexes are especially useful for testing the removal of an index before dropping it or using indexes temporarily without affecting the overall application.

## **What is poor Indexing?**

Any SQL Server table configuration where performance suffers due to excessive, improper, or missing indexes is considered to be poor indexing. If indexes are not properly created, SQL Server has to go through more records in order to retrieve the data requested by a query. Therefore, it uses more hardware resources (processor, memory, disk, and network) and obtaining the data lasts longer.

A wrong index can be an index created on a column that doesn’t provide easier data manipulation or an index created on multiple columns which instead of speeding up queries, slows them down. A table without a clustered index can also be considered as a poor indexing practice. Execution of a SELECT statement, inserting, updating, and deleting records is in most cases slower on a heap table than on a clustered one.

## **Disadvantages of using Indexes:**

* Take additional disk space.
* INSERT, UPDATE and DELETE becomes slower because on each operation the indexes must also be updated, but will speed up UPDATE, SELECT if the WHERE condition has an indexed field. For clustered indexes, the time increase is more significant, as the records have to maintain the correct order in data pages. Whether a new record is inserted, or an existing deleted or updated, this usually requires the records to be reordered.
* Wrong indexes can significantly slow down SQL Server performance. But even the indexes that provide better performance for some operations, can add overhead for others.

## **Advantages of using Indexes:**

* Speed up SELECT query. Used to speed up data search and SQL query performance. The database indexes reduce the number of data pages that have to be read in order to find the specific record.
* Helps to make a row unique or without duplicates (primary, unique)
* If index is set to fill-text index, then we can search against large string values. For example, to find a word from a sentence etc.

| **Type** | **Description** |
| --- | --- |
| Clustered | A clustered index sorts and stores the data rows of the table or view in order based on the index key. This type of index is implemented as a B-tree structure that supports fast retrieval of the rows, based on their key values. |
| Non-clustered | A non-clustered index can be defined on a table or view with a clustered index or on a heap. Each row in the index contains the key value and a row locator. This locator points to the data row in the clustered index or heap having the key value. The rows in the index are stored in the order of the key values, but the data rows are not guaranteed to be in any particular order unless they are in a clustered index. |
| Unique | A unique index ensures that the key contains no duplicate values and therefore every row in the table or view is in some way unique. |
| Index with included columns | A non-clustered index that is extended to include non-key columns in addition to the key columns. |
| Full-text | A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server. It provides efficient support for sophisticated word searches in character string data. |
| Spatial | A spatial index provides the ability to perform certain operations more efficiently on spatial objects (spatial data) in a column of the geometry data type. The spatial index reduces the number of objects on which relatively costly spatial operations need to be applied. |
| Filtered | An optimized non-clustered index, especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes. |
| XML | A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the xml data type column. |

## **When indexes should be avoided?**

* Indexes should not be used on small tables.
* Indexes should not be used on columns that contain a high number of NULL values.
* Columns that are frequently manipulated or updated should not be indexed. Maintenance on the index can become excessive.
* The columns are not often used as a condition in the query
* Indexes should not be used on columns that return a high percentage of data rows when used as a filter condition in a query's WHERE clause. For instance, you would not have an entry for the word "the" or "and" in the index of a book.
* Tables that have frequent, large batch update jobs run can be indexed. However, the batch job's performance is slowed considerably by the index. The conflict of having an index on a table that is frequently loaded or manipulated by a large batch process can be corrected by dropping the index before the batch job, and then re-creating the index after the job has completed.

**When should indexes be created –**

* A column contains a wide range of values
* A column does not contain a large number of null values
* One or more columns are frequently used together in a where clause or a join condition

## **Why does index require maintenance?**

* As data is inserted, updated, and deleted from tables, indexes can become fragmented over time, which can increase the amount of disk I/O that is required to retrieve data.
* In addition, as data is modified, the distribution of the data often changes over time. This can affect how the query optimizer uses indexes when creating optimal execution plans.
* A side effect of changing data is changing index and column statistics. In order for the query optimizer to be able to make the optimal choice when creating an executing plan, it needs index and column statistics that are representative of the data and that are up-to-date.
* Over time, it not uncommon for new queries to be run against your data. It is very likely that your current indexing scheme may not best meet the index needs of these new queries, and new indexes may be needed.
* In other cases, indexes may no longer be used, and if this is the case, then they should be removed to reduce the overhead of maintaining them.

**Below are the key indexes maintenance points:**

1. Identify and remove index fragmentation
2. Identify skewed and outdated index and column statistics and ensure they are representative and up-to-date
3. Identify and create missing indexes
4. Identify and remove unused indexes
5. Creating and monitoring index maintenance jobs

## **Indexing Methods**

### **Ordered Indices:**

The indices are usually sorted so that the searching is faster. The indices which are sorted are known as ordered indices.

* If the search key of any index specifies same order as the sequential order of the file, it is known as primary index or clustering index.  
  **Note:** The search key of a primary index is usually the primary key, but it is not necessarily so.
* If the search key of any index specifies an order different from the sequential order of the file, it is called the secondary index or non-clustering index.

### **Unique Index**

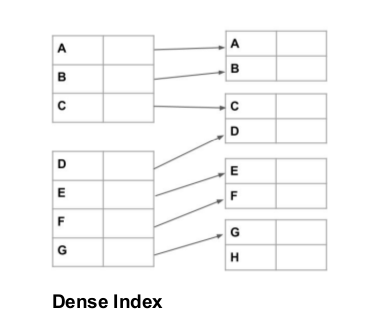
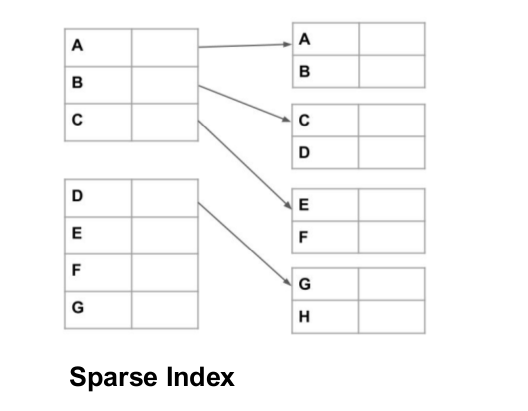
For creating a unique index, the user has to check the data in the column because the unique indexes are used when any column of the table has unique values. This indexing does not allow the field to have duplicate values if the column is unique indexed. A unique index can be applied automatically when a primary key is defined.

### **Primary Index :**

In this case, the data is sorted according to the search key. It induces sequential file organization. In this case, the primary key of the database table is used to create the index. As primary keys are unique and are stored in sorted manner, the performance of searching operation is quite efficient. The primary index is classified into two types: **Dense Index** and **Sparse Index**.

1. **Dense Index :**

* For every search key value in the data file, there is an index record.
* This record contains the search key and also a reference to the first data record with that search key value.

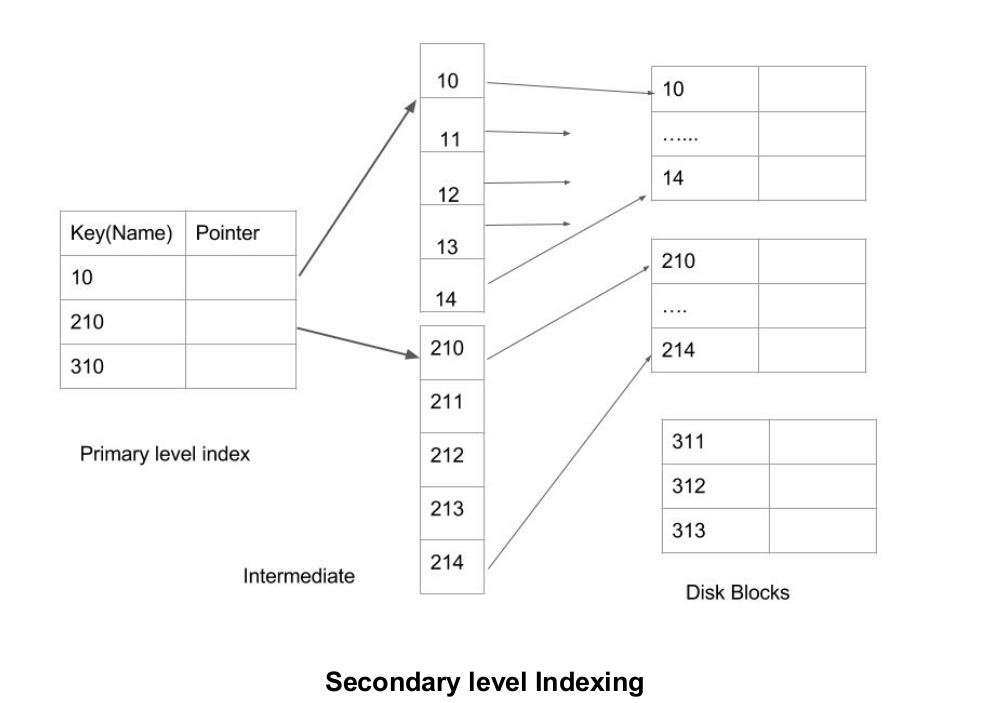
1. **Sparse Index:**

* The index record appears only for a few items in the data file. Each item points to a block as shown.
* To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
* We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.

### **Secondary Index :**

It is used to optimize query processing and access records in a database with some information other than the usual search key (primary key). In this, two levels of indexing are used in order to reduce the mapping size of the first level and in general. Initially, for the first level, a large range of numbers is selected so that the mapping size is small. Further, each range is divided into further sub ranges.

In order for quick memory access, first level is stored in the primary memory. Actual physical location of the data is determined by the second mapping level.



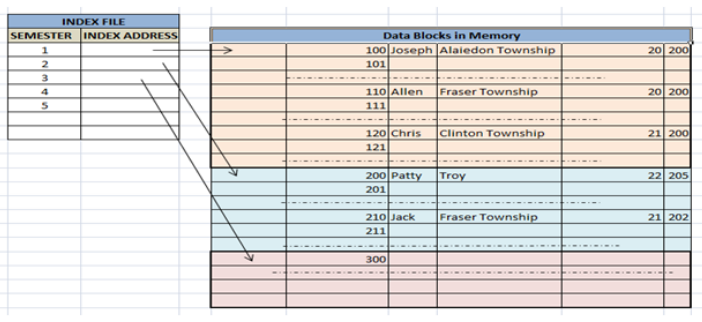
### **Clustered index:**

The clustered index is used to reorder the physical order of the table and search based on the key values. Each table can have only one clustered index. The Clustered index is the only index which has been automatically created when the primary key is generated. If moderate data modification needed to be done in the table then clustered indexes are preferred.

Indexing strategy is complex; it depends on many factors, including database structure, queries, and stored procedures used. One of general recommendations is to create a clustered index on tables where data is frequently queried. Although some DBAs and developers don’t prefer having clustered indexes on tables frequently inserted or updated, others consider that a clustered index on the right column can improve performance in these situations.

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc are grouped.



**Clustered index sorted according to first name (Search key)**

**Clustered Index:**

A clustered index affects how the data is actually stored. In a *heap*, the data rows are stored in no particular order. They are written wherever they fit while putting the minimum load on SQL Server resources, such as the buffer pool and the I/O subsystem. On the other hand, when you create a *clustered index* on a table, the organization of the data is changed so that it is now in order according to the keys specified. The entire index is organized as a B-tree (“B” stands for balanced), where the leaf nodes are the actual data pages and one or more levels of index nodes are built on top of the leaf nodes up to the single, root node. The result provides some guarantees regarding the asymptotic performance of the index. Most operations (search, insert and delete) operate in O (log *n*) where *n* is the number of entries in the index.

The clustered index is used to reorder the physical order of the table and search based on the key values. Each table can have only one clustered index. The Clustered index is the only index which has been automatically created when the primary key is generated. If moderate data modification needed to be done in the table then clustered indexes are preferred. With a proper clustered index, less reads are required to retrieve the records requested by a query or stored procedure. Therefore, fewer disk I/O are preformed and the operation is completed faster.

CREATE TABLE [Person].[Address](

    [AddressID] [int] IDENTITY(1,1) NOT FOR REPLICATION NOT NULL,

    [AddressLine1] [nvarchar](60) NOT NULL,

    [AddressLine2] [nvarchar](60) NULL,

    [City] [nvarchar](30) NOT NULL,

CONSTRAINT [PK\_Address\_AddressID] PRIMARY KEY CLUSTERED

(

    [AddressID] ASC

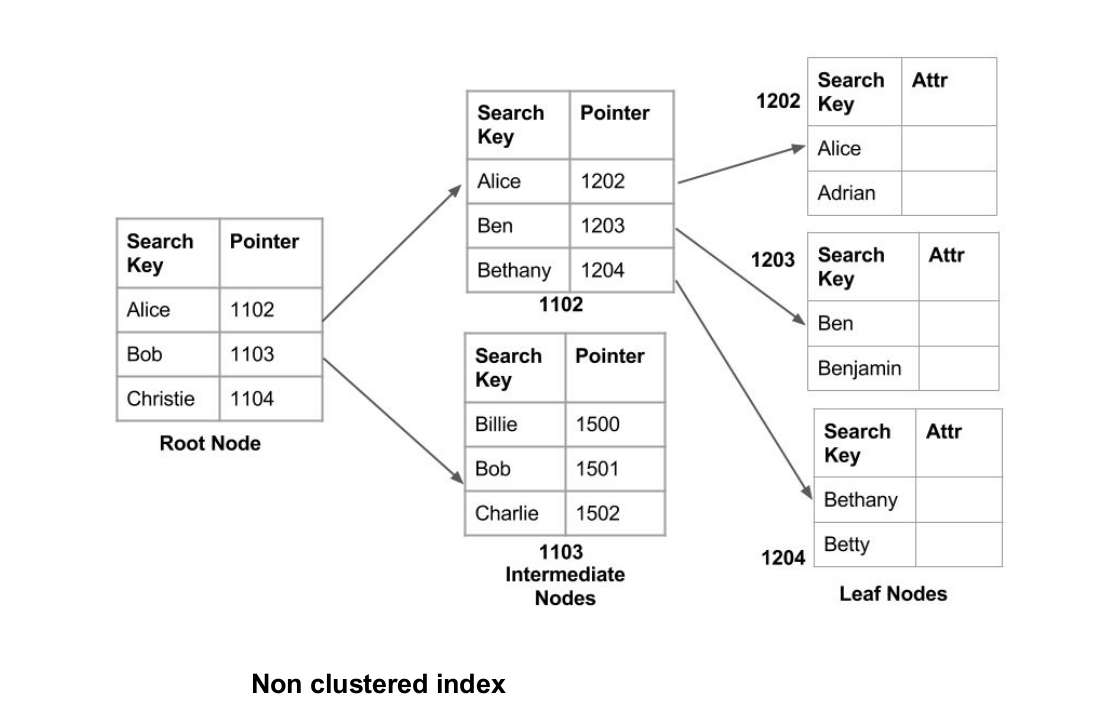
) ON [PRIMARY]

When you execute the select statement on a clustered table where an ascending clustered index is created, the results will be ordered ascending by the clustered key column.

### **Non-clustered index:**

The reason to create non-clustered index is searching the data. We well know that clustered indexes are created automatically when primary keys are generated, but non-clustered indexes are created when multiple joins conditions and various filters are used in the query. Non-Clustered Index does not alter the physical order of the table and maintains logical order of data. Each table can have 999 non-clustered indexes.

A non-clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here (information on each page of book) is not organized but we have an ordered reference (contents page) to where the data points actually lie. It requires more time as compared to clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In case of clustered index, data is directly present in front of the index.



A non-clustered index shares the B-tree concept for the index nodes with the same performance guarantees. However, such indexes do *not* affect the organization of the data pages, which may be clustered or not. Some optional features of nonclustered indexes are:

* Unique indexes – where the index entries must be unique and SQL Server makes sure that they are
* Filtered indexes – which are indexes built with a WHERE clause to limit what gets included in the index
* Included columns – which can carry a subset of non-key columns as part of the index.

The reason to create non-clustered index is searching the data. We well know that clustered indexes are created automatically primary keys are generated, but non-clustered indexes are created when multiple joins conditions and various filters are used in the query. Non-Clustered Index does not alter the physical order of the table and maintains logical order of data. Each table can have 999 non-clustered indexes. SQL Server allows up to 16 columns in an unclustered index.

CREATE TABLE [Person].[Address4](

    [AddressID] [int] IDENTITY(1,1) NOT FOR REPLICATION NOT NULL,

    [AddressLine1] [nvarchar](60) NOT NULL,

    [AddressLine2] [nvarchar](60) NULL,

    [City] [nvarchar](30) NOT NULL)

CREATE NONCLUSTERED INDEX [IX\_Address\_StateProvinceID4] ON [Person].[Address4]

(

    [AddressID] ASC

) ON [PRIMARY]

When you execute the select statement on a heap table with the same columns and data, the results returned will be unordered.

### **Bit-Map Index**

If Table contains the distinct values which are not more than 20 distinct values then user should go for Bit map indexes. User should avoid the indexing on each and every row and do the indexing only on distinct records of the table column. You should able to check drastic change in query cost after changing the normal index to Bit map index. The bit map indexes are very much useful in data warehousing where there are low level of concurrent transactions. Bit map index stores row\_id as associated key value with bitmap and did the indexing only distinct values. Means If in 1 million records only 20 distinct values are there so Bitmap index only stores 20 values as bitmap and fetches the records from that 20 values only.

CREATE BITMAP index  BM\_DEPT\_NAME on DEPT(Department\_name);

### **Composite Index**

An index that is created on more than one column is called “composite index”. A composite index, also called a concatenated index, is an index on multiple columns in a table. Columns in a composite index should appear in the order that makes the most sense for the queries that will retrieve data and need not be adjacent in the table.

When 2 or more columns in single table are related which each other and used in where condition of select statement then user should create composite index on the columns which are created. If all columns selected by in query are in composite index then oracle will return the values from the index without accessing the table. **Composite indexes should be avoided as they are large in size.**

Composite indexes can speed retrieval of data for SELECT statements in which the WHERE clause references all or the leading portion of the columns in the composite index. *Therefore, the order of the columns used in the definition is important.* In general, the most commonly accessed columns go first.

Suppose we want to do indexing on Employee and Department table. Here in above example Empno and Deptno are related to each other. So, we can create index on Employee\_Num and Department\_Num.

### **Covering Index**

Includes all the columns that are used in a particular query (or set of queries), allowing the database to use only the index and not actually have to look at the table data to retrieve the results.

### **Filtered Index**

The primary reason that we use filtered indexes is to address a query or set of queries that only requires a small portion of a table in order to return the data requested. This is common in tables where status exist that delineate active data from complete or archived data. It is also common when we have very narrow search terms that consistently address a very selective data set, such as orders that are not yet shipped, employees that are sick, or products that are out of stock.

CREATE NONCLUSTERED INDEX FI\_Employee\_DOJ

ON Employee(DOJ)

WHERE DOJ IS NOT NULL --Notice here the filter criteria for the index

A Filtered Index, which is an optimized non-clustered index, allows us to define a filter predicate, a WHERE clause, while creating the index. The B-Tree containing rows from the filtered index will contain only those rows which satisfy the filter criteria used while creating the index. This optimized index offers several benefits over a full table non-clustered index as follows:

* As discussed above, the filtered index contains only those rows which satisfy the defined filter criteria. As a result, it reduces the storage space requirement for the index.
* The filtered statistics or statistics for a filtered index are more compact and accurate, because they consider only the rows in the filtered index and the smaller size of the filtered index reduces the cost/overhead of updating the statistics.
* The impact of data modification is less with a filtered index as it is updated only when the data of the index is impacted, or a new record is inserted matching the filter criteria.
* Maintenance costs will be reduced as well since only a subset of rows will be in consideration while re-organizing or rebuilding the index.
* And most important, as it is an optimized non-clustered index, the query performance will improve if only a subset of data, which is covered by the filtered index criteria, is required.

Whereas the benefits provided by all indexes are to improve read performance on specific columns or sets of columns for important queries, the cost paid for this is in write performance. With standard indexes, whenever a write operation is performed that happens to change columns included in an index, additional write operations need to be completed in order to update the index. As long as we properly maintain our indexes and don’t allow the number of them to get out of control, the read-write tradeoff we accept for those indexes is likely an excellent deal for us.

Filtered indexes contain additional metadata in the form of a WHERE clause, which can reference one or many columns, as illustrated in our previous examples. In addition to updating any sorting or include columns, the columns in the WHERE clause also participate in write operations when needed. This additional cost, when not accounted for, can result in unexpected latency or contention as SQL Server needs to check additional data values before updating the filtered index.

* **Improved query performance and plan quality**

A well-designed filtered index improves query performance and execution plan quality because it is smaller than a full-table non clustered index and has filtered statistics. The filtered statistics are more accurate than full-table statistics because they cover only the rows in the filtered index.

* **Reduced index maintenance costs**

An index is maintained only when data manipulation language (DML) statements affect the data in the index. A filtered index reduces index maintenance costs compared with a full-table nonclustered index because it is smaller and is only maintained when the data in the index is changed. It is possible to have a large number of filtered indexes, especially when they contain data that is changed infrequently. Similarly, if a filtered index contains only the frequently modified data, the smaller size of the index reduces the cost of updating the statistics.

* **Reduced index storage costs**

Creating a filtered index can reduce disk storage for nonclustered indexes when a full-table index is not necessary. You can replace a full-table nonclustered index with multiple filtered indexes without significantly increasing the storage requirements.

**Limitations and Restrictions**

* You cannot create a filtered index on a view. However, the query optimizer can benefit from a filtered index defined on a table that is referenced in a view. The query optimizer considers a filtered index for a query that selects from a view if the query results will be correct.
* You cannot create a filtered index on a table when the column accessed in the filter expression is of a CLR data type.
* Filtered indexes are defined on one table and only support simple comparison operators. If you need a filter expression that references multiple tables or has complex logic, you should create a view.
* A column in the filtered index expression does not need to be a key or included column in the filtered index definition if the filtered index expression is equivalent to the query predicate and the query does not return the column in the filtered index expression with the query results.
* A column in the filtered index expression should be a key or included column in the filtered index definition if the query predicate uses the column in a comparison that is not equivalent to the filtered index expression.
* A column in the filtered index expression should be a key or included column in the filtered index definition if the column is in the query result set.
* The clustered index key of the table does not need to be a key or included column in the filtered index definition. The clustered index key is automatically included in all non-clustered indexes, including filtered indexes.
* If the comparison operator specified in the filtered index expression of the filtered index results in an implicit or explicit data conversion, an error will occur if the conversion occurs on the left side of a comparison operator. A solution is to write the filtered index expression with the data conversion operator (CAST or CONVERT) on the right side of the comparison operator.

**Filtered indexes have the following advantages over indexed views:**

1. Reduced index maintenance costs. For example, the query processor uses fewer CPU resources to update a filtered index than an indexed view.
2. Improved plan quality. For example, during query compilation, the query optimizer considers using a filtered index in more situations than the equivalent indexed view.
3. Online index rebuilds. You can rebuild filtered indexes while they are available for queries. Online index rebuilds are not supported for indexed views. For more information, see the REBUILD option for [ALTER INDEX (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-index-transact-sql?view=sql-server-2017).
4. Non-unique indexes. Filtered indexes can be non-unique, whereas indexed views must be unique.

<https://www.sqlshack.com/filtered-indexes-performance-analysis-and-hidden-costs/>

| **Criteria** | **Filtered Index** | **Indexed Views** |
| --- | --- | --- |
| Only One Table | A Filtered Index is created on  column(s) of a particular table. | Index Views can be created on column(s) from multiple base tables. |
| Simple WHERE criteria | A Filtered Index cannot use complex logic in its WHERE clause, for example the LIKE clause is not allowed, only simple comparison operators are allowed. | This limitation does not apply to indexed views and you can design your criteria as complex as you want. |
| Can do Online Rebuild | A Filtered Index can be rebuilt online. | Indexed views cannot be rebuilt online. |
| Non-Unique or Unique | You can create your Filtered Index as a non-unique index. | Indexed views can only be created as unique index. |

### **Included Columns in Index**

Non-Clustered index is created by adding key columns that are restricted in the number, type and size of these columns. To overcome these restrictions in the index keys, you could add non-key columns when creating a non-clustered index, which are the Included Columns. The Included columns option is only available to the non-clustered index and not available to the clustered indexes.

A column cannot be involved as key and non-key in the same index. It is either a key column or a non-key, included column. The main difference between the key and non-key columns is in the way it is stored in the index. The key column stored in all the levels of the index B-tree structure, where the non-key column stored in the leaf level of the B-tree structure only.

Included columns can be varchar (max), nvarchar(max), varbinary(max) or XML data types, that you cannot add it as index keys. Computed columns can also be used as included columns.

You should take into consideration that adding these large data types as non-key columns will increase the disk space requirements, as the column values will be copied into the index leaf level in addition to the table or clustered index.

On the other hand, you still can’t use TEXT, NTEXT and IMAGE as included columns.

Once the non-clustered index created, you can’t drop any index non-key column unless you drop the index first. Also these non-key columns can’t be changed except changing it from NOT NULL to NULL or increasing the length of **varchar**, **nvarchar**, or **varbinary** columns.

<https://www.sqlshack.com/sql-server-non-clustered-indexes-with-included-columns/>

### **Global Index and Local Index**

Usually when you create index on the table has indexed but when you are using partitioned table we need to change the syntax of the create index and need to use the Global index for one to many relationship. Global index is one to many relationships which allows index partition to map to many table partitions. The global index can be partitioned by range or hash method and it can be defined on any kind of partitioned or non-partitioned table.

Local indexes are indexes where there is one to one mapping between index partition and table partition. These indexes are basically used to improve the performance of partitioned tables. Local indexes directly uses divide and conquer approach to generate the Fast and best execution plan of SQL Query.

### **B-Tree Index**

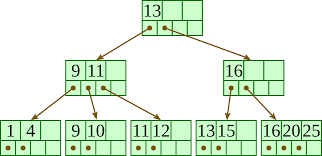
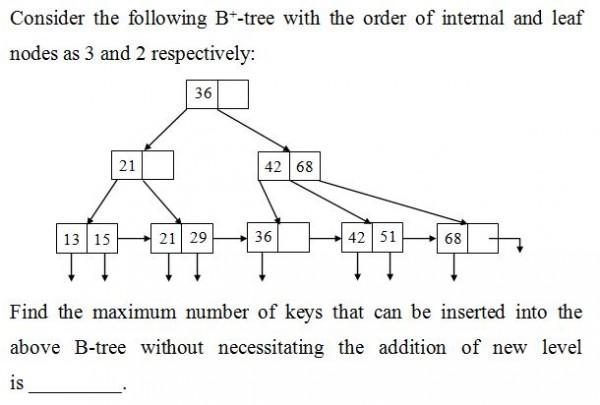
B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like [AVL](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/) and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. A B-tree index is an ordered list of values divided into ranges. By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches. A B-tree index has two types of blocks: **branch blocks** for searching and **leaf blocks** that store values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks. In [Figure 3-1](https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#i5765), the root branch block has an entry 0-40, which points to the leftmost block in the next branch level. This branch block contains entries such as 0-10 and 11-19. Each of these entries points to a leaf block that contains key values that fall in the range.

A B-tree index is balanced because all leaf blocks automatically stay at the same depth. Thus, retrieval of any record from anywhere in the index takes approximately the same amount of time. The **height** of the index is the number of blocks required to go from the root block to a leaf block. The **branch level** is the height minus 1.

Branch blocks store the minimum key prefix needed to make a branching decision between two keys. This technique enables the database to fit as much data as possible on each branch block. The branch blocks contain a pointer to the child block containing the key. The number of keys and pointers is limited by the block size.

The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row. Each entry is sorted by (key, rowid). Within a leaf block, a key and rowid is linked to its left and right sibling entries. The leaf blocks themselves are also doubly linked. In [Figure 3-1](https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#i5765) the leftmost leaf block (0-10) is linked to the second leaf block (11-19).

**Properties of B-Tree**  
**1)** All leaves are at same level.  
**2)** A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.  
**3)** Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.  
**4)** All nodes (including root) may contain at most 2t – 1 keys.  
**5)** Number of children of a node is equal to the number of keys in it plus 1.  
**6)** All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.  
**7)** B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.  
**8)** Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

### **Spatial Index**

A *spatial index* is a type of extended index that allows you to index a spatial column. A spatial column is a table column that contains data of a spatial data type, such as **geometry** or **geography**. As was the case with an XML index, spatial indexes also require that the database table that you are creating the spatial index on also has a clustered primary key index defined. In SQL Server, spatial indexes are built using B-trees, which means that the indexes must represent the 2-dimensional spatial data in the linear order of B-trees. Therefore, before reading data into a spatial index, SQL Server implements a hierarchical uniform decomposition of space. The index-creation process *decomposes* the space into a four-level *grid hierarchy*. These levels are referred to as *level 1* (the top level), *level 2*, *level 3*, and *level 4*.

Each successive level further decomposes the level above it, so each upper-level cell contains a complete grid at the next level. On a given level, all the grids have the same number of cells along both axes (for example, 4x4 or 8x8), and the cells are all one size.

When it comes to spatial data you are generally testing if two points and/or areas intersect or are within a certain distance.  The benefit of having a spatial index created on your column is it allows the query to easily prune/skip over column values that there is no chance of intersection.  When creating the index there are options available that increase the accuracy of the index but with that comes the drawback that the index will use more space.

CREATE SPATIAL INDEX IX\_Address\_SpatialLocation ON Person.Address(SpatialLocation);

### **Column Store Index**

The columnstore index is also logically organized as a table with rows and columns, but the data is physically stored in a column-wise data format. Columnstore indexes work well for mostly read-only queries with large data sets, like data warehousing workloads. Columnstore indexes are not well-suited for queries that seek specific individual values. Columns often contain similar data which enables the data to be highly compressed, improving memory utilization and significantly reducing disk usage. A columnstore index can be clustered or non-clustered. Clustered and non-clustered columnstore indexes function the same. The difference is that a clustered columnstore index provides the primary storage for the entire table, while a nonclustered index is a secondary index that contains a copy of some of the columns in the underlying table. A clustered columnstore index can have one or more non-clustered B-tree indexes.

It is a column-based non-clustered index geared toward increasing query performance for workloads that involve large amounts of data, typically found in data warehouse fact tables. Columnstore indexes are the standard for storing and querying large data warehousing fact tables. This index uses column-based data storage and query processing to achieve gains up to **10 times the query performance** in your data warehouse over traditional row-oriented storage. You can also achieve gains up to **10 times the data compression** over the uncompressed data size. Beginning with SQL Server 2016 (13.x), columnstore indexes enable operational analytics: the ability to run performant real-time analytics on a transactional workload. For analytics, a columnstore index offers an order of magnitude better performance than a btree index. Columnstore indexes are the preferred data storage format for data warehousing and analytics workloads.

This new type of index stores data column-wise instead of row-wise, as indexes currently do. Performance advantages in columnstore indexes are possible by leveraging the VertiPaq compression technology, which enables large amounts of data to be compressed in-memory. This in-memory compressed store reduces the number of disk reads and increases buffer cache hit ratios because only the smaller column-based data pages that need to satisfy a query are moved into memory.

For wide tables, such as those commonly found in data warehouses, columnstore indexes come in handy as you essentially reduce the amount and size of data needed to be accessed for any given query.

Reasons why columnstore indexes are so fast:

* Columns store values from the same domain and commonly have similar values, which result in high compression rates. I/O bottlenecks in your system are minimized or eliminated, and memory footprint is reduced significantly.
* High compression rates improve query performance by using a smaller in-memory footprint. In turn, query performance can improve because SQL Server can perform more query and data operations in memory.
* Batch execution improves query performance, typically by two to four times, by processing multiple rows together.
* Queries often select only a few columns from a table, which reduces total I/O from the physical media.

Recommended use cases:

* Use a clustered columnstore index to store fact tables and large dimension tables for data warehousing workloads. This method improves query performance and data compression by up to 10 times. For more information, see [Columnstore indexes for data warehousing](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-data-warehouse?view=sql-server-2017).
* Use a non-clustered columnstore index to perform analysis in real time on an OLTP workload. For more information, see [Get started with columnstore for real-time operational analytics](https://docs.microsoft.com/en-us/sql/relational-databases/indexes/get-started-with-columnstore-for-real-time-operational-analytics?view=sql-server-2017).

### **Row Store Index**

The rowstore index is the traditional style that has been around since the initial release of SQL Server. Rowstore indexes are designed to speed the retrieval of data by enabling queries to quickly locate data by index rather than scanning an entire table. Rowstore data is logically organized by rows and columns, and is physically stored in row-oriented data pages. SQL Server internally organizes rowstore indexes using a B-Tree structure over the data pages. SQL Server supports clustered and nonclustered indexes. With clustered indexes, the data in the base table is organized according to the clustered index. Rowstore indexes perform best on queries that seek data by searching for a particular value or retrieving a small range of values. Rowstore indexes are a good fit for transactional workloads since these workloads tend to require table seeks instead of large-range table scans, and they often require frequent data updates.

## **When to Use Clustered or Non-Clustered Indexes**

Now that you know the differences between a clustered and a non-clustered index, let’s see the different scenarios for using each of them.

1.   Number of Indexes

This is pretty obvious. If you need to create multiple indexes on your database, go for non-clustered index since there can be only one clustered index.

2.   SELECT Operations

If you want to select only the index value that is used to create and index, non-clustered indexes are faster. For example, if you have created an index on the “name” column and you want to select only the name, non-clustered indexes will quickly return the name.

However, if you want to select other column values such as age, gender using the name index, the SELECT operation will be slower since first the name will be searched from the index and then the reference to the actual table record will be used to search the age and gender.

On the other hand, with clustered indexes since all the records are already sorted, the SELECT operation is faster if the data is being selected from columns other than the column with clustered index.

3.   INSERT/UPDATE Operations

The INSERT and UPDATE operations are faster with non-clustered indexes since the actual records are not required to be sorted when an INSERT or UPDATE operation is performed. Rather only the non-clustered index needs updating.

4.   Disk Space

Since, non-clustered indexes are stored at a separate location than the original table, non-clustered indexes consume additional disk space. If disk space is a problem, use a clustered index.

5.   Final Verdict

As a rule of thumb, every table should have at least one clustered index preferably on the column that is used for SELECTING records and contains unique values. The primary key column is an ideal candidate for a clustered index.

On the other hand columns that are often involved in INSERT and UPDATE queries should have a non-clustered index assuming that disk space isn’t a concern.

## **Indexed views**

To enhance the performance of complex queries, a unique clustered index can be created on the view, where the result set of that view will be stored in your database the same as a real table with a unique clustered index. The good thing here is – the queries that are using the table itself can benefits from the view’s clustered index without calling the view itself. Maintaining the clustered index of the view to be unique, the data changed on the source table will be easily found and the change will be reflected to the view. Changing the data directly from the indexed view is possible but shouldn’t be done. Also, it is possible to create non-clustered indexes on a view, providing more possibilities to enhance the queries calling the view.

You can benefit from indexed views if its data is not frequently updated, as the performance degradation of maintaining the data changes of the indexed view is higher than the performance enhancement of using this Indexed View. Indexed views improve the performance of queries that use joins and aggregations in processing huge amount of data and are executed very frequently. The environments that are best suited to indexed views are data warehouses and the Online Analytical Processing (OLAP) databases.

It is important for indexed view that view must be created with *schemabinding* option. Schemabinding option ensure that the underlying tables cannot be altered in any way that would materially affect the indexed view unless the view is first altered or dropped. If we try to create an indexed view without schemabinding option then SQL Server will throw an error of “Cannot create index on view 'VW\_Employee' because the view is not schema bound”.

There are some limitations when you create an indexed view. You can’t use **EXISTS**, **NOT** **EXISTS**, **OUTER JOIN**, **COUNT(\*)**, **MIN**, **MAX**, **subqueries, table hints**, **TOP** and **UNION** in the definition of your indexed view.  Also, it is not allowed to refer to other views and tables in other databases in the view definition. You can’t use the **text**, **ntext**, **image**and **XML**, data types in your indexed views. **Float** data type can be used in the indexed view but can’t be used in the clustered index. If the Indexed view’s definition contains **GROUP BY** clause, you should add **COUNT\_BIG(\*)** to the view definition

**Where to Use Indexed View**

Indexed views have both a benefit and a cost. The benefit is that query optimizer provides more efficient and faster results for complex and redundant queries. The cost of an indexed view is on the maintenance of the clustered index. In the following scenario, Indexed View can be used.

* When you use the same complex query on many tables, multiple times.
* When new system need to read old table data, but doesn't watch to change their perceived schema.
* The environments that are best suited for indexed views are data warehouses, data marts, OLAP databases but transactional environment are less suitable for Indexed View.

**Restrictions on Indexed Views**

* View must be created with SCHEMABINDING.
* Functions using in definition of view must have been created with SCHEMABINDING.
* Base tables must have been created with the proper ANSI\_NULLS setting.

## **What is RAW datatype**

In Oracle PL/SQL, **RAW** is a data type used to store binary data, or data which is byte oriented (for example, graphics or audio files). One of the most important things to note about RAW data is that *it can only be queried or inserted*; RAW data cannot be manipulated. RAW data is always returned as a hexadecimal character value. In SQL, its maximum size is 2000 bytes, while in PL/SQL it is 32767.

## **CTE**

A common table expression (CTE) is a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, APPLY or CREATE VIEW statement. A CTE is similar to a derived table in that it is not stored as an object and lasts only for the duration of the query. Unlike a derived table, a CTE can be self-referencing and can be referenced multiple times in the same query. A CTE always returns a result set.  They are used to simplify queries, for example, you could use one to eliminate a derived table from the main query body.

A **CTE** can be used:  
• For recursion  
• Substitute for a view when the general use of a view is not required; that is, you do not have to store the definition in metadata.  
• Reference the resulting non-large table multiple times in the same statement.

WITH TableExpressionName (Column1, Column2, …, ColumnN)

AS

(Query Definition)

**Guidelines for creating a CTE**

* A CTE must be followed by a single SELECT, INSERT, UPDATE, or DELETE statement that references some or all the CTE columns. A CTE can also be specified in a CREATE VIEW statement as part of the defining SELECT statement of the view.
* Multiple CTE query definitions can be defined in a non-recursive CTE. The definitions must be combined by one of these set operators: UNION ALL, UNION, INTERSECT, or EXCEPT.
* A CTE can reference itself and previously defined CTEs in the same WITH clause. Forward referencing is not allowed.
* Specifying more than one WITH clause in a CTE is not allowed. For example, if a *CTE\_query\_definition* contains a subquery, that subquery cannot contain a nested WITH clause that defines another CTE.
* The following clauses cannot be used in the *CTE\_query\_definition*:
* ORDER BY (except when a TOP clause is specified)
* INTO
* OPTION clause with query hints
* FOR BROWSE
* When a CTE is used in a statement that is part of a batch, the statement before it must be followed by a semicolon.
* A query referencing a CTE can be used to define a cursor.
* Tables on remote servers can be referenced in the CTE.
* When executing a CTE, any hints that reference a CTE may conflict with other hints that are discovered when the CTE accesses its underlying tables, in the same manner as hints that reference views in queries. When this occurs, the query returns an error.

**Non-Recursive CTE:**

There are many reasons to use non-recursive common table expressions.  They include:

* **Readability** – Non-recursive CTEs can make your query easier to read by organizing complex code, and eliminating the need to repeat complicated expressions.
* **Substitute for a View** – Views are great for encapsulating query logic and promoting reuse, but there are times when you you’re either unable to create a view due to permissions, or the view would only be used in one query.
* **Limitations** – Overcome SELECT statement limitations, such as referencing itself (recursion), or performing GROUP BY using non-deterministic functions.
* **Ranking** – Whenever you want to use ranking function such as ROW\_NUMBER(), RANK(), NTILE() etc.

**Recursive CTE:**

Recursive CTEs are special in the sense they are allowed to reference themselves. Because of this special ability, you can use recursive CTEs to solve problems other queries cannot. A recursive CTE is a CTE that references itself.  In doing so, the initial CTE is repeatedly executed, returning subsets of data, until the complete result is returned. A recursive CTE must contain a UNION ALL statement and, to be recursive, have a second query definition which references the CTE itself unlike the non-recursive CTE.

Being able to reference itself is a unique feature and benefit.  It allows recursive CTE’s to solve queries problems that would otherwise require the use of temporary tables, cursors, and other means. Recursive CTE’s are well suited to querying hierarchical data, such as organization charts or production bill of materials, where each product’s components are made up of subcomponents, and so on.

**The following guidelines apply to defining a recursive common table expression:**

* The recursive CTE definition must contain at least two CTE query definitions, an anchor member and a recursive member. Multiple anchor members and recursive members can be defined; however, all anchor member query definitions must be put before the first recursive member definition. All CTE query definitions are anchor members unless they reference the CTE itself.
* Anchor members must be combined by one of these set operators: UNION ALL, UNION, INTERSECT, or EXCEPT. UNION ALL is the only set operator allowed between the last anchor member and first recursive member, and when combining multiple recursive members.
* The number of columns and their corresponding data types should be the same between the anchor and recursive members. This makes sense, as this is a condition for UNION ALL.
* The FROM clause of a recursive member must refer only one time to the CTE *expression\_name*.
* The following items are not allowed in the *CTE\_query\_definition* of a recursive member:
  + SELECT DISTINCT
  + GROUP BY
  + PIVOT (When the database compatibility level is 110 or higher
  + HAVING
  + Scalar aggregation
  + TOP
  + LEFT, RIGHT, OUTER JOIN (INNER JOIN is allowed)
  + Subqueries
  + A hint applied to a recursive reference to a CTE inside a CTE\_query\_definition.

## **Stored Procedure**

A stored procedure is a way for you to store a set of SQL statements and accompanying programming statements within the database and run them later. Stored procedures come in handy as they allow you combine both procedural logic as well as SQL statements. This makes SQL Server very flexible, as SQL by itself isn’t suitable to tackle all problems, and being able to call upon procedural logic to string together multiple SQL statements into step is handy. For those familiar with the concept of a programming, a procedure is a set of instructions used to perform an action.  In databases, this is taken a step further; database stored procedures, are more specialized and built to perform specific database tasks such as inserting or deleting data.

**Benefits of Stored Procedure:**

1. **Security**: Since stored procedures are stored within the database you can set up security to restrict which users are able see and execute them.  This is handy if you have sensitive queries or code which you want to run in a controlled manner. Not only can you control who can see the stored procedure code, and perhaps sensitive information contained in the queries themselves, but you can control access to who can execute them.
2. **Isolation of business rule**
3. **Maintainability**: Stored procedures make it easier to maintain complicated code as they promote [modular programming](https://www.techopedia.com/definition/25972/modular-programming). Keep in mind that stored procedures can call other stored procedures.  This is important, it allows you to take a very complicated process, and break into down into manageable pieces.
4. **Speed/Optimization:** Stored procedures generally run faster than individual statements called from client programs. They reduce network traffic since you’re sending one command to execute the stored procedure over the wire rather than several queries to execute. Also, in addition stored procedures are cached on the server, so once executed, the queries, are ready for repeated execution.
5. **Testing**: When you place business rules within stored procedures, you have an opportunity to test them independently of the of any applications you’re building.
6. Stored procedures help in reducing the network traffic and latency. It boosts up the application performance.
7. Stored procedures facilitate the reusability of the code.
8. You can encapsulate the logic using stored procedures and change stored procedure code without affecting clients.
9. It is possible to reuse stored procedure execution plans, which are cached in SQL Server's memory. This reduces server overhead.
10. It provides modularity of application.

**To call a stored procedure with output parameters, follow these steps:**

* First, declare [variables](http://www.sqlservertutorial.net/sql-server-stored-procedures/variables/) to hold the value returned by the output parameters
* Second, use these variables in the stored procedure call.

DECLARE @count INT;

EXEC uspFindProductByModel

    @model\_year = 2018,

    @product\_count = @count OUTPUT;

SELECT @count AS 'Number of products found';

## **User Defined Functions**

We write the function for reusability purposes to avoid complexity and duplication of code. User-defined functions are precompiled and ready for later use. Using SQL server, you can easily create, view, and modify the user-defined functions. Functions are computed values and cannot perform permanent environmental changes to SQL Server (i.e., no INSERT or UPDATE statements allowed). A function can be used inline in SQL statements if it returns a scalar value or can be joined upon if it returns a result set. SQL Server user-defined functions are routines that accept the parameters and perform the action such as complex calculation and return the result of that action as value. User-defined scalar functions allow you to encapsulate complex formula or business logic and reuse them in every query.

* The function accepts only input parameters
* The functions don’t support the exceptional handling

**Benefits of user-defined functions**

* It supports modular programming
* Create once and reuse any number of times
* Once compiled, same execution plan we can reuse, It reduces the compilation cost
* It reduces the network traffic
* We can use a user-defined function in select as well as where class.
* We can use a user-defined function used as rawest.
* We can use a user-defined function as parameterizing views.

In SQL Server there are three types of user-defined functions

1. *Scalar-Valued Function*The function which returns the single value. It may or may not have the parameters. The scalar functions help you simplify your code. For example, you may have a complex calculation that appears in many [queries](http://www.sqlservertutorial.net/sql-server-basics/sql-server-select/). Instead of including the formula in every query, you can create a scalar function that encapsulates the formula and uses it in the queries. Scalar functions can use logic such as [IF](http://www.sqlservertutorial.net/sql-server-stored-procedures/sql-server-if-else/) blocks or [WHILE](http://www.sqlservertutorial.net/sql-server-stored-procedures/sql-server-while/) loops. They functions cannot [update](http://www.sqlservertutorial.net/sql-server-basics/sql-server-update/) data. They can access data but this is not a good practice. Scalar functions can call other functions.
2. *Inline table-valued function*An inline table-valued function returns a row set of SQL server data type and specifies only the table keyword in the return clause, without table definition information.  
   The code inside the functions returns a single return statement that invokes the select statement.
3. *Multi-Statement table valued function*This function returns the table variable as a result of an action performed by function and table variable should be explicitly declared and defined by whose value can be derived from multiple SQL statements.

## **Table Valued Functions**

A table-valued function is a [user-defined function](http://www.sqlservertutorial.net/sql-server-user-defined-functions/) that returns data of a table type. The return type of a table-valued function is a table, therefore, you can use the table-valued function just like you would use a table. Think of inline table valued functions as views that accept parameters. Inline table-valued function and multi-statement table-valued function are user-defined functions that return data of table types. We typically use table-valued functions as parameterized [views](http://www.sqlservertutorial.net/sql-server-views/). In comparison with [stored procedures](http://www.sqlservertutorial.net/sql-server-stored-procedures/), the table-valued functions are more flexible because we can use them wherever tables are used.

In SQL server you have three kinds of user-defined functions\*: scalar functions (svf), multi-line table valued functions (mTVF) and inline table valued functions (iTVF). svfs return a single value, both mTVFs and iTVFs return a table. The difference between mTVFs and iTVFs is performance. In short - mTVFs are slow, iTVFs can be (and almost always are) much faster. mTVFs allow you to do things you couldn't do in a view (e.g. create temp tables, perform loops, utilize cursors...), iTVFs, again, have the same restrictions as views except for they can except parameters.

I use iTFVs for common data warehouse queries where I need a view that takes parameter and splitting/manipulating strings. The function which returns the result set of a single [SELECT](http://www.sqlservertutorial.net/sql-server-basics/sql-server-select/) statement is known as an inline table-valued function.

To execute a table-valued function, you use it in the FROM clause of the SELECT statement:

SELECT     \* FROM     udfProductInYear(2017);

A multi-statement table-valued function or MSTVF is a table-valued function that returns the result of multiple statements. The multi-statement-table-valued function is very useful because you can execute multiple queries within the function and aggregate results into the returned table. To define a multi-statement table-valued function, you use a table variable as the return value. Inside the function, you execute one or more queries and insert data into this table variable.

## **Table Variables**

Table variables are kind of variables that allow you to hold rows of data, which are similar to a [temporary tables](http://www.sqlservertutorial.net/sql-server-basics/sql-server-temporary-tables/).

DECLARE @table\_variable\_name TABLE (    column\_list );

**Restrictions**

1. We have to define the structure of the table variable during the declaration. Unlike a regular or temporary table, you cannot [alter](http://www.sqlservertutorial.net/sql-server-basics/sql-server-alter-table-alter-column/) the structure of the table variable after the declaration.
2. Statistics help the query optimizer to come up with a good query’s execution plan. Unfortunately, table variables do not contain statistics. Therefore, you should use table variables to hold a small number of rows.
3. We cannot use the table variable as an [input](http://www.sqlservertutorial.net/sql-server-stored-procedures/stored-procedure-output-parameters/) or [output parameter](http://www.sqlservertutorial.net/sql-server-stored-procedures/stored-procedure-output-parameters/) like other [data types](http://www.sqlservertutorial.net/sql-server-basics/sql-server-data-types/). However, you can return a table variable from a user-defined function
4. We cannot create non-clustered indexes for table variables. However, starting with SQL Server 2014, memory-optimized table variables are available with the introduction of the new In-Memory OLTP that allows you to add non-clustered indexes as part of table variable’s declaration.
5. If we are using a table variable with a [join](http://www.sqlservertutorial.net/sql-server-basics/sql-server-inner-join/), you need to [alias](http://www.sqlservertutorial.net/sql-server-basics/sql-server-alias/) the table in order to execute the query.

**Performance**

Using table variables in a [stored procedure](http://www.sqlservertutorial.net/sql-server-stored-procedures/) results in fewer recompilations than using a [temporary table](http://www.sqlservertutorial.net/sql-server-basics/sql-server-temporary-tables/). In addition, a table variable use fewer resources than a temporary table with less locking and logging overhead. Similar to the temporary table, the table variables do live in the tempdb database, not in the memory.

## **Triggers**

A trigger is a special type of stored procedure that automatically runs when an event occurs in the database server. DML triggers run when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view. These triggers fire when any valid event fires, whether table rows are affected or not. A trigger is designed to check or change data based on a data modification or definition statement; it should't return data to the user. DML triggers are frequently used for enforcing business rules and data integrity.

DDL triggers run in response to a variety of data definition language (DDL) events. These events primarily correspond to Transact-SQL CREATE, ALTER, and DROP statements, and certain system stored procedures that perform DDL-like operations. Unlike DML triggers, DDL triggers aren't scoped to schemas. So, you can't use functions such as OBJECT\_ID, OBJECT\_NAME, OBJECTPROPERTY, and OBJECTPROPERTYEX for querying metadata about DDL triggers. Use the catalog views instead.

Logon triggers fire in response to the LOGON event that's raised when a user's session is being established. Logon triggers don't fire if authentication fails. You can create triggers directly from Transact-SQL statements or from methods of assemblies that are created in the Microsoft .NET Framework common language runtime (CLR) and uploaded to an instance of SQL Server. SQL Server lets you create multiple triggers for any specific statement.

Triggers work in transactions (implied or otherwise) and while they're open, they lock resources. The lock remains in place until the transaction is confirmed (with COMMIT) or rejected (with a ROLLBACK). The longer a trigger runs, the higher the probability that another process is then blocked. So, write triggers to lessen their duration whenever possible. One way to achieve shorter duration is to release a trigger when a DML statement changes zero rows.

SQL Server also supports recursive invocation of triggers when the RECURSIVE\_TRIGGERS setting is enabled using ALTER DATABASE.

Recursive triggers enable the following types of recursion to occur:

* **Indirect recursion**: With indirect recursion, an application updates table T1. This fires trigger TR1, updating table T2. Trigger T2 then fires and updates table T1.
* **Direct recursion:** In direct recursion, the application updates table T1. This fires trigger TR1, updating table T1. Because table T1 was updated, trigger TR1 fires again, and so on.

You can nest triggers to a maximum of 32 levels. If a trigger changes a table on which there's another trigger, the second trigger activates and can then call a third trigger, and so on. If any trigger in the chain sets off an infinite loop, the nesting level is exceeded and the trigger is canceled. When a Transact-SQL trigger launches managed code by referencing a CLR routine, type, or aggregate, this reference counts as one level against the 32-level nesting limit. Methods invoked from within managed code don't count against this limit.

To disable nested triggers, set the nested triggers option of sp\_configure to 0 (off). The default configuration supports nested triggers. If nested triggers are off, recursive triggers are also disabled, despite the RECURSIVE\_TRIGGERS setting that's set by using ALTER DATABASE.

The first AFTER trigger nested inside an INSTEAD OF trigger fires even if the **nested triggers** server configuration option is 0. But, under this setting, the later AFTER triggers don't fire. Review your applications for nested triggers to determine if the applications follow your business rules when the **nested triggers** server configuration option is set to 0. If not, make the appropriate modifications.

**Limitations of Triggers:**

* CREATE TRIGGER must be the first statement in the batch and can apply to only one table.
* A trigger is created only in the current database; however, a trigger can reference objects outside the current database.
* If the trigger schema name is specified to qualify the trigger, qualify the table name in the same way.
* The same trigger action can be defined for more than one user action (for example, INSERT and UPDATE) in the same CREATE TRIGGER statement.
* INSTEAD OF DELETE/UPDATE triggers can't be defined on a table that has a foreign key with a cascade on DELETE/UPDATE action defined.
* Any SET statement can be specified inside a trigger. The SET option selected remains in effect during the execution of the trigger and then reverts to its former setting.
* When a trigger fires, results are returned to the calling application, just like with stored procedures. To prevent results being returned to an application because of a trigger firing, don't include either SELECT statements that return results or statements that carry out variable assignment in a trigger. A trigger that includes either SELECT statements that return results to the user or statements that do variable assignment, requires special handling. You'd have to write the returned results into every application in which modifications to the trigger table are allowed. If variable assignment must occur in a trigger, use a SET NOCOUNT statement at the start of the trigger to prevent the return of any result sets.

**DDL Trigger :** The DDL triggers are fired in response to DDL (Data Definition Language) command events that start with Create, Alter and Drop, such as Create\_table, Create\_view, drop\_table, Drop\_view and Alter\_table.

1. **create** **trigger** saftey
2. **on** **database**
3. **for**
4. create\_table,alter\_table,drop\_table
5. **as**
6. print 'you can not create ,drop and alter table in this database'
7. **rollback**;

The DDL triggers are useful in the following cases:

* Record changes in the database schema.
* Prevent some specific changes to the database schema.
* Respond to a change in the database schema.

**DML Trigger :** The DML triggeres are fired in response to DML (Data Manipulation Language) command events that start with with Insert, Update and Delete. Like insert\_table, Update\_view and Delete\_table.

1. **create** **trigger** deep
2. **on** emp
3. **for**
4. **insert**,**update**,**delete**
5. **as**
6. print'you can not insert,update and delete this table i'
7. **rollback**;

**INSTEAD OF Trigger :** It will tell the database engine to execute the trigger instead of executing the statement. For example, an insert trigger executes when an event occurs instead of the statement that would insert the values in the table.

1. **CREATE** **TRIGGER** instoftr
2. **ON** v11
3. **INSTEAD** **OF** **INSERT**
4. **AS**
5. **BEGIN**
6. **INSERT** **INTO** emp
7. **SELECT** I.id, I.names
8. **FROM** INSERTED I

**CREATE TRIGGER:**

The CREATE TRIGGER statement allows you to create a new trigger that is fired automatically whenever an event such as [INSERT](http://www.sqlservertutorial.net/sql-server-basics/sql-server-insert/), [DELETE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-delete/), or [UPDATE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-update/) occurs against a table.

CREATE TRIGGER [schema\_name.]trigger\_name

ON table\_name

AFTER  {[INSERT],[UPDATE],[DELETE]}

[NOT FOR REPLICATION]

AS

{sql\_statements}

**INSTEAD OF:**

An INSTEAD OF trigger is a trigger that allows you to skip an [INSERT](http://www.sqlservertutorial.net/sql-server-basics/sql-server-insert/), [DELETE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-delete/), or [UPDATE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-update/) statement to a table or a view and execute other statements defined in the trigger instead. The actual insert, delete, or update operation does not occur at all.

In other words, an INSTEAD OF trigger skips a DML statement and execute other statement.

CREATE TRIGGER [schema\_name.] trigger\_name

ON {table\_name | view\_name }

INSTEAD OF {[INSERT] [,] [UPDATE] [,] [DELETE] }

AS

{sql\_statements}

**DISABLE TRIGGER :**

Sometimes, for the troubleshooting or data recovering purpose, you may want to disable a trigger temporarily.

DISABLE TRIGGER [schema\_name.][trigger\_name]

ON [object\_name | DATABASE | ALL SERVER]

**ENABLE TRIGGER :**

The ENABLE TRIGGER statement allows you to enable a trigger so that the trigger can be fired whenever an event occurs.

ENABLE TRIGGER [schema\_name.][trigger\_name]

ON [object\_name | DATABASE | ALL SERVER]

**TO VIEW A TRIGGER :**

You can get the definition of a trigger by querying data against the sys.sql\_modules view:

SELECT

    definition

FROM

    sys.sql\_modules

WHERE

    object\_id = OBJECT\_ID('sales.trg\_members\_delete');

You can get the definition of a trigger using the OBJECT\_DEFINITION function as follows:

SELECT

    OBJECT\_DEFINITION (

        OBJECT\_ID(

            'sales.trg\_members\_delete'

        )

    ) AS trigger\_definition;

EXEC sp\_helptext 'sales.trg\_members\_delete' ;

**DROP TRIGGER :**

The SQL Server DROP TRIGGER statement drops one or more triggers from the database.

DROP TRIGGER [ IF EXISTS ] [schema\_name.]trigger\_name [ ,...n ];

To remove one or more [DDL triggers](http://www.sqlservertutorial.net/sql-server-triggers/sql-server-ddl-trigger/), you use the following form of the DROP TRIGGER statement:

DROP TRIGGER [ IF EXISTS ] trigger\_name [ ,...n ]

ON { DATABASE | ALL SERVER };

**TO LIST TRIGGERS :**

To list all triggers in a SQL Server, you query data from the sys.triggers view:

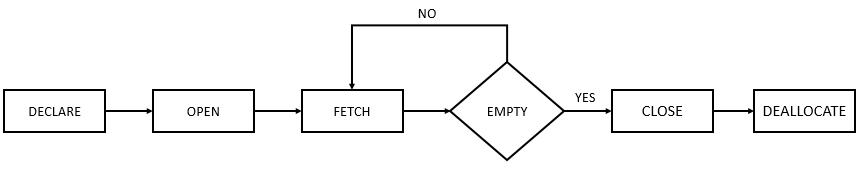
SELECT  name,    is\_instead\_of\_trigger

FROM     sys.triggers

WHERE     type = 'TR';

## **Cursor**

A database cursor is an object that enables traversal over the rows of a result set. It allows you to process individual row returned by a query. Sometimes the application logic needs to work with a row at a time rather than the entire result set at once. In T-SQL, one way of doing this is using a CURSOR. These are steps for using a cursor:



In the cursor execution, we have two steps. Step one, the positioning, when the cursor sets its position to a row from the result set. Step two, the retrieval, when it gets the data from that specific row in an operation called the FETCH. Finally, CLOSE syntax releases the current result set and removes the locks from the rows used by the cursor, and DEALLOCATE removes cursor reference.

SQL Server cursors are notoriously bad for performance because SQL Server, like any good relational database management system (RDBMS), is optimized for set-based operations.

## **Transaction**

A [Database Transaction](http://en.wikipedia.org/wiki/Database_transaction) is a set of database operations that must be treated as whole, means either all operations are executed or none of them. An example can be bank transaction from one account to another account. Either both debit and credit operations must be executed or none of them. [ACID](http://en.wikipedia.org/wiki/ACID)(Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.

Transaction is the smallest stack of the process which cannot be divided into smaller pieces. Also, some group of transaction process can be performed sequentially but as we explained in the Atomicity principle if even one of the transactions fails, all transaction blocks will fail. It is the responsibility of an enterprise database system, such as an instance of the Database Engine, to provide mechanisms ensuring the physical integrity of each transaction. The Database Engine provides:

* Locking facilities that preserve transaction isolation.
* Logging facilities that ensure transaction durability. Even if the server hardware, operating system, or the instance of the Database Engine itself fails, the instance uses the transaction logs upon restart to automatically roll back any uncompleted transactions to the point of the system failure.

Transaction management features that enforce transaction atomicity and consistency. After a transaction has started, it must be successfully completed (committed), or the Database Engine undoes all of the data modifications made since the transaction started. This operation is referred to as rolling back a transaction because it returns the data to the state it was prior to those changes

**SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;** for the seat and booking update. As Serializable level guarantees safety from Dirty, Nonrepeatable and Phantoms reads.

"ReadCommitted" + Optimistic/Pessimistic Lock should work just fine. Without lock, DB transaction itself doesn't cover the complete transaction semantics.

The SQL Server Database Engine uses the following mechanisms to ensure the integrity of transactions and maintain the consistency of databases when multiple users are accessing data at the same time:

* **Locking**

Each transaction requests locks of different types on the resources, such as rows, pages, or tables, on which the transaction is dependent. The locks block other transactions from modifying the resources in a way that would cause problems for the transaction requesting the lock. Each transaction frees its locks when it no longer has a dependency on the locked resources.

* **Row versioning**

When a row versioning-based isolation level is enabled, the Database Engine maintains versions of each row that is modified. Applications can specify that a transaction use the row versions to view data as it existed at the start of the transaction or query instead of protecting all reads with locks. By using row versioning, the chance that a read operation will block other transactions is greatly reduced.

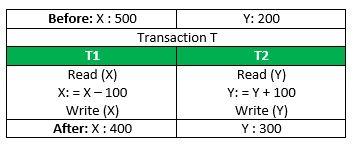
Locking and row versioning prevent users from reading uncommitted data and prevent multiple users from attempting to change the same data at the same time. Without locking or row versioning, queries executed against that data could produce unexpected results by returning data that has not yet been committed in the database.

## **ACID Properties of a transaction**

ACID properties are some basic rules, which has to be satisfied by every transaction to preserve the integrity. A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations. The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

These properties and rules are:

1. **Atomicity:** Atomicity is more generally known as all or nothing rule. Which implies all transactions are considered as one unit, and they either run to completion or not executed at all. Either the entire transaction is performed or the transaction returns to the beginning. That is, all changes made by the transaction are undone and returned to their previous state. A transaction block cannot be left unattended. Half of the remaining transaction block causes data inconsistency. Transactions do not occur partially. A debit transaction should either execute all operations or none. If debit transaction fails in middle after executing any operation, then its new value will not be updated in the database which leads to inconsistency. It reflects the principle of indivisibility that we describe as the main feature of the transaction process. A transaction involves following two operations:  
   —**Abort**: If a transaction aborts, changes made to database are not visible.  
   —**Commit**: If a transaction commits, changes made are visible.  
   Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account X to account Y.



If the transaction fails after completion of **T1** but before completion of **T2**.( say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

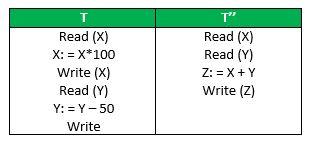
1. **Consistency:** This property refers to the uniformity of the data. Consistency implies that the database is consistent before and after the transaction. If operations of debit and credit transactions on same account are executed concurrently, it may leave database in an inconsistent state. There is a rule that sets the substructure of the non-divisibility rule. Transaction data must provide consistency. That is, if the update operation is performed in a transaction, either all remaining transactions must be performed, or the update operation must be canceled. This data is very important in terms of consistency. This means that integrity constraints must be maintained so that the database is consistent before and after the transaction.

* For Example, T1 (debit of Rs. 1000 from A) and T2 (credit of 500 to A) executing concurrently, the database reaches inconsistent state.
* Let us assume Account balance of A is Rs. 5000. T1 reads A(5000) and stores the value in its local buffer space. Then T2 reads A(5000) and also stores the value in its local buffer space.
* T1 performs A=A-1000 (5000-1000=4000) and 4000 is stored in T1 buffer space. Then T2 performs A=A+500 (5000+500=5500) and 5500 is stored in T2 buffer space. T1 writes the value from its buffer back to database.
* A’s value is updated to 4000 in database and then T2 writes the value from its buffer back to database. A’s value is updated to 5500 which shows that the effect of debit transaction is lost and database has become inconsistent.
* To maintain consistency of database, we need **concurrency control protocols**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

1. **Isolation:** This property states that the number of the transaction can be executed concurrently without leading to the inconsistency of the database state. Result of a transaction should not be visible to others before transaction is committed. Each transaction must be processed separately. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

For example, let us assume that A’s balance is Rs. 5000 and T1 debits Rs. 1000 from A. A’s new balance will be 4000. If T2 credits Rs. 500 to A’s new balance, A will become 4500 and after this T1 fails. Then we have to rollback T2 as well because it is using value produced by T1. So, a transaction results are not made visible to other transactions before it commits.

Let **X**=500, **Y** =500.  
Consider two transactions **T** and **T”.**



Suppose **T** has been executed till **Read (Y)** and then **T’’** starts. As a result , interleaving of operations takes place due to which **T’’** reads correct value of **X** but incorrect value of **Y** and sum computed by  
**T’’: (X+Y = 50, 000+500=50, 500)**  
is thus not consistent with the sum at end of transaction:  **T: (X+Y = 50, 000 + 450 = 50, 450)**.  
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after a they have been made to the main memory.

1. **Durable:** This property ensures that once the transaction is committed it will be stored in the non-volatile memory and system crash can also not affect it anymore. Once database has committed a transaction, the changes made by the transaction should be permanent. Transactions can perform complex operations with data. In order to secure all these transactions, they must be resistant to a transaction error. System problems that may occur in SQL Server should be prepared and resilient against power failure, operating system or other software-induced errors. This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if system failure occurs. These updates now become permanent and are stored in a non-volatile memory. The effects of the transaction, thus, are never lost.

e.g.; If a person has credited $500000 to his account, bank can’t say that the update has been lost. To avoid this problem, multiple copies of database are stored at different locations.

## **Transaction Types**

* **Explicit Transactions**  
  An explicit transaction is one in which you explicitly define both the start and end of the transaction through an API function or by issuing the Transact-SQL BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK Transact-SQL statements. When the transaction ends, the connection returns to the transaction mode it was in before the explicit transaction was started, either implicit or autocommit mode. Explicit transactions are explicitly controlled by the code. You can start them by using **BEGIN TRAN** statement. They will remain active until you explicitly call **COMMIT** or **ROLLBACK** in the code.

You can use all Transact-SQL statements in an explicit transaction, except for the following statements:

| ALTER DATABASE | CREATE DATABASE | DROP FULLTEXT INDEX |
| --- | --- | --- |
| ALTER FULLTEXT CATALOG | CREATE FULLTEXT CATALOG | RECONFIGURE |
| ALTER FULLTEXT INDEX | CREATE FULLTEXT INDEX | RESTORE |
| BACKUP | DROP DATABASE | Full-text system stored procedures |
| CREATE DATABASE | DROP FULLTEXT CATALOG | sp\_dboption to set database options or any system procedure that modifies the master database inside explicit or implicit transactions. |

**Note**

UPDATE STATISTICS can be used inside an explicit transaction. However, UPDATE STATISTICS commits independently of the enclosing transaction and cannot be rolled back.

* **Autocommit Transactions**  
  Autocommit mode is the default transaction management mode of the SQL Server Database Engine. Every Transact-SQL statement is committed or rolled back when it completes. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back. A connection to an instance of the Database Engine operates in autocommit mode whenever this default mode has not been overridden by either explicit or implicit transactions. Autocommit mode is also the default mode for ADO, OLE DB, ODBC, and DB-Library. In case, when there are no active transactions present, SQL Server would use autocommittedtransactions – starting transactions and committing them for each statement it executes. Autocommitted transactions work on per-statement rather than per-module level. For example, when a stored procedure consists of five statements; SQL Server would have five autocommitted transactions executed. Moreover, if this procedure failed in the middle of execution, SQL Server would not roll back previously committed autocommitted transactions. This behavior may lead to logical data inconsistency in the system.
* **Implicit Transactions**  
  When a connection is operating in implicit transaction mode, the instance of the Database Engine automatically starts a new transaction after the current transaction is committed or rolled back. You do nothing to delineate the start of a transaction; you only commit or roll back each transaction. Implicit transaction mode generates a continuous chain of transactions. Set implicit transaction mode on through either an API function or the Transact-SQL SET IMPLICIT\_TRANSACTIONS ON statement. For the logic that includes multiple data modification statements, autocommitted transactions are less efficient than explicit transactions due to the logging overhead they introduce. In this mode, every statement would generate transaction log records for implicit **BEGIN TRAN** and **COMMIT** operations, which leads to the large amount of transaction log activity and degrade performance of the system. SQL Server also supports implicit transactions, which you can enable with **SET IMPLICIT\_TRANSACTION ON** statement. When this option is enabled, SQL Server starts the new transaction when there is no active explicit transactions present. This transaction stays active until you explicitly issue **COMMIT** or **ROLLBACK** statement. Implicit transactions may make transaction management more complicated and they are rarely used in production. However, there is the caveat – **SET ANSI\_DEFAULT ON** option also automatically enables implicit transactions. This behavior may lead to unexpected concurrency issues in the system.

After implicit transaction mode has been set on for a connection, the instance of the Database Engine automatically starts a transaction when it first executes any of these statements:

| ALTER TABLE | FETCH | REVOKE |
| --- | --- | --- |
| CREATE | GRANT | SELECT |
| DELETE | INSERT | TRUNCATE TABLE |
| DROP | OPEN | UPDATE |

* **Batch-scoped Transactions**  
  Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.
* **Distributed Transactions**  
  Distributed transactions span two or more servers known as resource managers. The management of the transaction must be coordinated between the resource managers by a server component called a transaction manager. Each instance of the SQL Server Database Engine can operate as a resource manager in distributed transactions coordinated by transaction managers, such as Microsoft Distributed Transaction Coordinator (MS DTC), or other transaction managers that support the Open Group XA specification for distributed transaction processing. For more information, see the MS DTC documentation.

A transaction within a single instance of the Database Engine that spans two or more databases is actually a distributed transaction. The instance manages the distributed transaction internally; to the user, it operates as a local transaction.

At the application, a distributed transaction is managed much the same as a local transaction. At the end of the transaction, the application requests the transaction to be either committed or rolled back. A distributed commit must be managed differently by the transaction manager to minimize the risk that a network failure may result in some resource managers successfully committing while others roll back the transaction. This is achieved by managing the commit process in two phases (the prepare phase and the commit phase), which is known as a two-phase commit (2PC).

* + Prepare phase  
    When the transaction manager receives a commit request, it sends a prepare command to all of the resource managers involved in the transaction. Each resource manager then does everything required to make the transaction durable, and all buffers holding log images for the transaction are flushed to disk. As each resource manager completes the prepare phase, it returns success or failure of the prepare to the transaction manager.
  + Commit phase  
    If the transaction manager receives successful prepares from all of the resource managers, it sends commit commands to each resource manager. The resource managers can then complete the commit. If all of the resource managers report a successful commit, the transaction manager then sends a success notification to the application. If any resource manager reported a failure to prepare, the transaction manager sends a rollback command to each resource manager and indicates the failure of the commit to the application.

Database Engine applications can manage distributed transactions either through Transact-SQL or the database.

***Ending Transaction :***

You can end transactions with either a COMMIT or ROLLBACK statement, or through a corresponding API function.

* COMMIT  
  If a transaction is successful, commit it. A COMMIT statement guarantees all the transaction's modifications are made a permanent part of the database. A COMMIT also frees resources, such as locks, used by the transaction.
* ROLLBACK  
  If an error occurs in a transaction, or if the user decides to cancel the transaction, then roll the transaction back. A ROLLBACK statement backs out all modifications made in the transaction by returning the data to the state it was in at the start of the transaction. A ROLLBACK also frees resources held by the transaction.

***Errors during Transaction processing:***

If an error prevents the successful completion of a transaction, SQL Server automatically rolls back the transaction and frees all resources held by the transaction. If a run-time statement error (such as a constraint violation) occurs in a batch, the default behavior in the Database Engine is to roll back only the statement that generated the error. You can change this behavior using the SET XACT\_ABORT statement. After SET XACT\_ABORT ON is executed, any run-time statement error causes an automatic rollback of the current transaction. Compile errors, such as syntax errors, are not affected by SET XACT\_ABORT. When errors occur, corrective action (COMMIT or ROLLBACK) should be included in application code. One effective tool for handling errors, including those in transactions, is the Transact-SQL TRY…CATCH construct. A compile error prevents the Database Engine from building an execution plan, so nothing in the batch is executed.

## **Concurrency Control**

Concurrency Control deals with**interleaved execution** of more than one transaction.

**What is Transaction?**

A set of logically related operations is known as transaction. The main operations of a transaction are:

**Read(A):** Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in main memory.

**Write (A):** Write operation Write(A) or W(A) writes the value back to the database from buffer.

But it may also be possible that transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power** etc. For example, if debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:

**Commit:** After all instructions of a transaction are successfully executed, the changes made by transaction are made permanent in the database.

**Rollback:** If a transaction is not able to execute all operations successfully, all the changes made by transaction are undone.

Concurrency control theory has two classifications for the methods of instituting concurrency control:

* **Pessimistic concurrency control**

A system of locks prevents users from modifying data in a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control because it is mainly used in environments where there is high contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

* **Optimistic concurrency control**

In optimistic concurrency control, users do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called optimistic because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read.

***Concurrency Effects:***

Users modifying data can affect other users who are reading or modifying the same data at the same time. These users are said to be accessing the data concurrently. If a data storage system has no concurrency control, users could see the following side effects:

* *Lost updates*

Lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of the other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

For example, two editors make an electronic copy of the same document. Each editor changes the copy independently and then saves the changed copy thereby overwriting the original document. The editor who saves the changed copy last overwrites the changes made by the other editor. This problem could be avoided if one editor could not access the file until the other editor had finished and committed the transaction.

* *Uncommitted dependency (dirty read)*

Uncommitted dependency occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row.

For example, an editor is making changes to an electronic document. During the changes, a second editor takes a copy of the document that includes all the changes made so far, and distributes the document to the intended audience. The first editor then decides the changes made so far are wrong and removes the edits and saves the document. The distributed document contains edits that no longer exist and should be treated as if they never existed. This problem could be avoided if no one could read the changed document until the first editor does the final save of modifications and commits the transaction.

* *Inconsistent analysis (nonrepeatable read)*

Inconsistent analysis occurs when a second transaction accesses the same row several times and reads different data each time. Inconsistent analysis is similar to uncommitted dependency in that another transaction is changing the data that a second transaction is reading. However, in inconsistent analysis, the data read by the second transaction was committed by the transaction that made the change. Also, inconsistent analysis involves multiple reads (two or more) of the same row, and each time the information is changed by another transaction; thus, the term nonrepeatable read.

For example, an editor reads the same document twice, but between each reading the writer rewrites the document. When the editor reads the document for the second time, it has changed. The original read was not repeatable. This problem could be avoided if the writer could not change the document until the editor has finished reading it for the last time.

* *Phantom reads*

A phantom read is a situation that occurs when two identical queries are executed and the collection of rows returned by the second query is different. The example below shows how this may occur. Assume the two transactions below are executing at the same time. The two SELECT statements in the first transaction may return different results because the INSERT statement in the second transaction changes the data used by both.

Copy

--Transaction 1

BEGIN TRAN;

SELECT ID FROM dbo.employee

WHERE ID > 5 and ID < 10;

--The INSERT statement from the second transaction occurs here.

SELECT ID FROM dbo.employee

WHERE ID > 5 and ID < 10;

COMMIT;

Copy

--Transaction 2

BEGIN TRAN;

INSERT INTO dbo.employee

SET name = 'New' WHERE ID = 5;

COMMIT;

* *Missing and double reads caused by row updates*
  + Missing an updated row or seeing an updated row multiple times

Transactions that are running at the READ UNCOMMITTED level do not issue shared locks to prevent other transactions from modifying data read by the current transaction. Transactions that are running at the READ COMMITTED level do issue shared locks, but the row or page locks are released after the row is read. In either case, when you are scanning an index, if another user changes the index key column of the row during your read, the row might appear again if the key change moved the row to a position ahead of your scan. Similarly, the row might not appear if the key change moved the row to a position in the index that you had already read. To avoid this, use the SERIALIZABLE or HOLDLOCK hint, or row versioning. For more information, see [Table Hints (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table).

* + Missing one or more rows that were not the target of update

When you are using READ UNCOMMITTED, if your query reads rows using an allocation order scan (using IAM pages), you might miss rows if another transaction is causing a page split. This cannot occur when you are using read committed because a table lock is held during a page split and does not happen if the table does not have a clustered index, because updates do not cause page splits.

## **Isolation Level**

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Isolation levels are described in terms of which concurrency side effects, such as dirty reads or phantom reads, are allowed. Based on these phenomena, The **SQL** standard defines four **isolation levels** - **Uncommitted** **Read** (also called "dirty read"), **Committed Read**, **Repeatable Read**, and **Serializable**. Read Uncommitted is the lowest **isolation level**. Serializable is the Highest **isolation level**.

Transaction isolation levels control the following:

* Whether locks are taken when data is read, and what type of locks are requested.
* How long the read locks are held.
* Whether a read operation referencing rows modified by another transaction:
  + Block until the exclusive lock on the row is freed.
  + Retrieve the committed version of the row that existed at the time the statement or transaction started.
  + Read the uncommitted data modification.

Choosing a transaction isolation level doesn't affect the locks that are acquired to protect data modifications. A transaction always gets an exclusive lock on any data it modifies and holds that lock until the transaction completes, regardless of the isolation level set for that transaction. For read operations, transaction isolation levels primarily define the level of protection from the effects of modifications made by other transactions.

A lower isolation level increases the ability of many users to access data at the same time, but increases the number of concurrency effects, such as dirty reads or lost updates, that users might encounter. Conversely, a higher isolation level reduces the types of concurrency effects that users might encounter, but requires more system resources and increases the chances that one transaction will block another. Choosing the appropriate isolation level depends on balancing the data integrity requirements of the application against the overhead of each isolation level. The highest isolation level, serializable, guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation, but it does this by performing a level of locking that is likely to impact other users in multi-user systems. The lowest isolation level, read uncommitted, can retrieve data that has been modified but not committed by other transactions. All concurrency side effects can happen in read uncommitted, but there's no read locking or versioning, so overhead is minimized.

## **Transaction Isolation Level**

As we know that, in order to maintain consistency in a database, it follows ACID properties. Among these four properties (Atomicity, Consistency, Isolation and Durability) Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system. You can change the level of isolation that a particular connection is operating in by using the SET TRANSACTION ISOLATION LEVEL command.

SET TRANSACTION ISOLATION LEVEL

\[READ UNCOMMITTED | READ COMMITTED | REPEATABLE

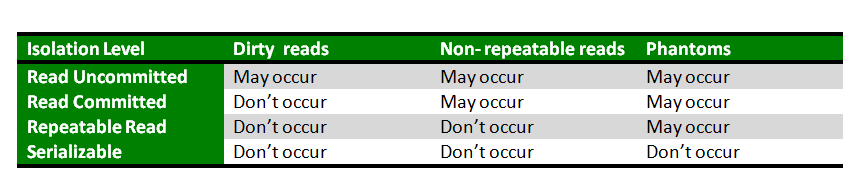
READ | SERIALIZABLE\]  
Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena –

* **Dirty Read –**A Dirty read is the situation when a transaction reads a data that has not yet been committed. For example, Let’s say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
* **Non Repeatable read –**Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, Now, if transaction T1 rereads the same data, it will retrieve a different value.
* **Phantom Read –**Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time.

Based on these phenomena, The SQL standard defines four isolation levels :

1. **Read Uncommitted –**Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.
2. **Read Committed –**This isolation level guarantees that any data read is committed at the moment it is read. Thus, it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.
3. **Repeatable Read –**This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **Serializable –**This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing. The Serializable isolation level ensures that if a query is reissued, no data will have changed and no new rows will appear in the interim. In other words, you won't see phantoms if the same query is issued twice within a transaction. Rerun the example from the Repeatable Reads section, inserting a row with a col1 value of 35. But this time, set your isolation level to SERIALIZABLE. The second connection will block when you try to do the INSERT, and the first connection will read exactly the same rows each time.

The Table is given below clearly depicts the relationship between isolation levels, read phenomena and locks :



## **Dirty Read**

When a transaction is allowed to read a row that has been modified by another transaction which is not committed yet that time, *Dirty Reads* occurred. It is mainly occurred because of multiple transaction at a time which is not committed.

If we have a ticket booking system and One Customer is trying to book a ticket at that time available number of the ticket is 10, before completing the payment, the Second Customer wants to book a ticket that time this 2nd transaction will show the second customer that the number of the available tickets is 9. The twist is here if the first customer does not have sufficient fund in his debit card or in his wallet then the 1st transaction will Rollback, that time 9 seat available which is read by the 2nd transaction is *Dirty Read*.

## **Row Versioning Isolation Level**

| **Row Versioning Isolation Level** | **Definition** |
| --- | --- |
| Read Committed Snapshot | When the READ\_COMMITTED\_SNAPSHOT database option is set ON, read committed isolation uses row versioning to provide statement-level read consistency. Read operations require only SCH-S table level locks and no page or row locks. That is, the Database Engine uses row versioning to present each statement with a transactionally consistent snapshot of the data as it existed at the start of the statement. Locks are not used to protect the data from updates by other transactions. A user-defined function can return data that was committed after the time the statement containing the UDF began.  When the READ\_COMMITTED\_SNAPSHOT database option is set OFF, which is the default setting, read committed isolation uses shared locks to prevent other transactions from modifying rows while the current transaction is running a read operation. The shared locks also block the statement from reading rows modified by other transactions until the other transaction is completed. Both implementations meet the ISO definition of read committed isolation. |
| Snapshot | The snapshot isolation level uses row versioning to provide transaction-level read consistency. Read operations acquire no page or row locks; only SCH-S table locks are acquired. When reading rows modified by another transaction, they retrieve the version of the row that existed when the transaction started. You can only use Snapshot isolation against a database when the ALLOW\_SNAPSHOT\_ISOLATION database option is set ON. By default, this option is set OFF for user databases.  **Note:** SQL Server does not support versioning of metadata. For this reason, there are restrictions on what DDL operations can be performed in an explicit transaction that is running under snapshot isolation. The following DDL statements are not permitted under snapshot isolation after a BEGIN TRANSACTION statement: ALTER TABLE, CREATE INDEX, CREATE XML INDEX, ALTER INDEX, DROP INDEX, DBCC REINDEX, ALTER PARTITION FUNCTION, ALTER PARTITION SCHEME, or any common language runtime (CLR) DDL statement. These statements are permitted when you are using snapshot isolation within implicit transactions. An implicit transaction, by definition, is a single statement that makes it possible to enforce the semantics of snapshot isolation, even with DDL statements. Violations of this principle can cause error 3961: "Snapshot isolation transaction failed in database '%.\*ls' because the object accessed by the statement has been modified by a DDL statement in another concurrent transaction since the start of this transaction. It is not allowed because the metadata is not versioned. A concurrent update to metadata could lead to inconsistency if mixed with snapshot isolation." |

## **Schedule**

A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A schedule is a series of operations from one or more transactions. A schedule can be of two types:

* **Serial Schedule:** When one transaction completely executes before starting another transaction, the schedule is called serial schedule. A serial schedule is always consistent. e.g.; If a schedule S has debit transaction T1 and credit transaction T2, possible serial schedules are T1 followed by T2 (T1->T2) or T2 followed by T1 ((T1->T2). A serial schedule has low throughput and less resource utilization.
* **Concurrent Schedule:** When operations of a transaction are interleaved with operations of other transactions of a schedule, the schedule is called Concurrent schedule. e.g.; Schedule of debit and credit transaction shown in Table 1 is concurrent in nature. But concurrency can lead to inconsistency in the database.  The above example of a concurrent schedule is also inconsistent.

## **Types of Schedule**

Types of schedules.

**1. Serial Schedules –**  
Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

**Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
| R(B) |  |
|  | W(B) |
|  | R(A) |
|  | R(B) |

where R(A) denotes that a read operation is performed on some data item ‘A’  
This is a serial schedule since the transactions perform serially in the order T1 —> T2

**2. Complete Schedules –**  
Schedules in which the last operation of each transaction is either abort (or) commit are called complete schedules.

**Example:** Consider the following schedule involving three transactions T1, T2 and T3.

| **T1** | **T2** | **T3** |
| --- | --- | --- |
| R(A) |  |  |
|  | W(A) |  |
| R(B) |  |  |
|  |  | W(B) |
| commit |  |  |
|  | commit |  |
|  |  | abort |

This is a complete schedule since the last operation performed under every transaction is either “commit” or “abort”.

**3. Recoverable Schedules –**  
Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction Tj is reading value updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.

**Example –** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
|  | R(A) |
| commit |  |
|  | Commit |

This is a recoverable schedule since T1 commits before T2, that makes the value read by T2 correct.

**Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
|  | R(A) |
|  | Commit |
| abort |  |

T2 read the value of A written by T1, and committed. T1 later aborted, therefore the value read by T2 is wrong, but since T2 committed, this schedule is **unrecoverable**.

**4. Cascadeless Schedules –**  
Also called Avoids cascading aborts/rollbacks (ACA). Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.

**Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
| commit |  |
|  | R(A) |
|  | commit |

This schedule is cascadeless. Since the updated value of **A** is read by T2 only after the updating transaction i.e. T1 commits.

**Example:** Consider the following schedule involving two transactions T1 and T2.

| **T1** | **T2** |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | R(A) |
|  | W(A) |
| abort |  |
|  | abort |

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T1aborts, T2 will have to be aborted too in order to maintain the correctness of the schedule as T2 has already read the uncommitted value written by T1.

**5. Strict Schedules –**  
A schedule is strict if for any two transactions Ti, Tj, if a write operation of Ti precedes a conflicting operation of Tj (either read or write), then the commit or abort event of Ti also precedes that conflicting operation of Tj.  
In other words, Tj can read or write updated or written value of Ti only after Ticommits/aborts.

**Example:** Consider the following schedule involving two transactions T1 and T2.

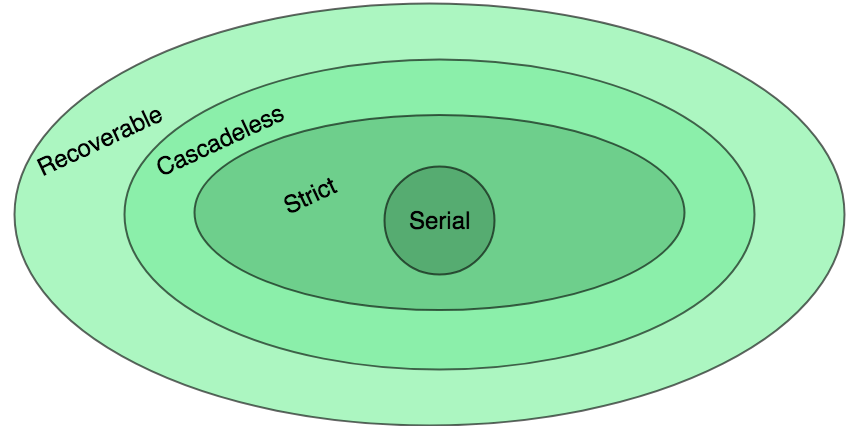
| **T1** | **T2** |
| --- | --- |
| R(A) |  |
|  | R(A) |
| W(A) |  |
| commit |  |
|  | W(A) |
|  | R(A) |
|  | commit |

This is a strict schedule since T2 reads and writes A which is written by T1 only after the commit of T1.

**Note –** It can be seen that:

1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

The relation between various types of schedules can be depicted as:



**Example:** Consider the following schedule:

S:R1(A), W2(A), Commit2, W1(A), W3(A), Commit3, Commit1

Which of the following is true?  
(A) The schedule is view serializable schedule and strict recoverable schedule  
(B) The schedule is non-serializable schedule and strict recoverable schedule  
(C) The schedule is non-serializable schedule and is not strict recoverable schedule.  
(D) The Schedule is serializable schedule and is not strict recoverable schedule

**Solution:** The schedule can be re-written as:-

| **T1** | **T2** | **T3** |
| --- | --- | --- |
| R(A) |  |  |
|  | W(A) |  |
|  | Commit |  |
| W(A) |  |  |
|  |  | W(A) |
|  |  | Commit |
| Commit |  |  |

First of all, it is a view serializable schedule as it has view equal serial schedule T1 —> T2 —> T3 which satisfies the initial and updated reads and final write on variable A which is required for view serializability. Now we can see there is write – write pair done by transactions T1 followed by T3 which is violating the above-mentioned condition of strict schedules as T3 is supposed to do write operation only after T1 commits which is violated in the given schedule. Hence the given schedule is serializable but not strict recoverable.  
So, option (D) is correct.

## **Conflict Serializability**

As discussed in [Concurrency control](http://quiz.geeksforgeeks.org/concurrency-control-introduction/) , serial schedules have less resource utilization and low throughput. To improve it, two or more transactions are run concurrently. But concurrency of transactions may lead to inconsistency in database. To avoid this, we need to check whether these concurrent schedules are serializable or not.

A Schedule is called **view serializable** if it is view equal to a serial schedule (no overlapping transactions).

**Conflict Serializable:** A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

**Conflicting operations:** Two operations are said to be conflicting if all conditions satisfy:

* They belong to different transactions
* They operate on the same data item
* At least one of them is a write operation

Example: –

* **Conflicting** operations pair (R1(A), W2(A)) because they belong to two different transactions on same data item A and one of them is write operation.
* Similarly, (W1(A), W2(A)) and (W1(A), R2(A)) pairs are also **conflicting**.
* On the other hand, (R1(A), W2(B)) pair is **non-conflicting** because they operate on different data item.
* Similarly, ((W1(A), W2(B)) pair is **non-conflicting.**

Consider the following schedule:

S1: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)

If Oi and Oj are two operations in a transaction and Oi< Oj(Oiis executed before Oj), same order will follow in schedule as well. Using this property, we can get two transactions of schedule S1 as:

T1: R1(A), W1(A), R1(B), W1(B)

T2: R2(A), W2(A), R2(B), W2(B)

**Possible Serial Schedules are: T1->T2 or T2->T1**

-> **Swapping non-conflicting operation**s R2(A) and R1(B) in S1, the schedule becomes,

**S11:** R1(A), W1(A), R1(B), **W2(A),** R2(A), **W1(B),** R2(B), W2(B)

-> Similarly, s**wapping non-conflicting operations** W2(A) and W1(B) in S11, the schedule becomes,

**S12:** R1(A), W1(A), R1(B), W1(B), R2(A), W2(A), R2(B), W2(B)

S12 is a serial schedule in which all operations of T1 are performed before starting any operation of T2. Since S has been transformed into a serial schedule S12 by swapping non-conflicting operations of S1, S1 is conflict serializable.

Let us take another Schedule:

S2: R2(A), W2(A), R1(A), W1(A), R1(B), W1(B), R2(B), W2(B)

Two transactions will be:

T1: R1(A), W1(A), R1(B), W1(B)

T2: R2(A), W2(A), R2(B), W2(B)

**Possible Serial Schedules are: T1->T2 or T2->T1**

Original Schedule is:

**S2:** R2(A), W2(A), **R1(A),** W1(A), R1(B), W1(B), **R2(B),** W2(B)

Swapping non-conflicting operations R1(A) and R2(B) in S2, the schedule becomes,

**S21:** R2(A), W2(A), R2(B), **W1(A),** R1(B), W1(B), R1(A), **W2(B)**

Similarly, swapping non-conflicting operations W1(A) and W2(B) in S21, the schedule becomes,

**S22:** R2(A), W2(A), R2(B), W2(B), R1(B), W1(B), R1(A), W1(A)

In schedule S22, all operations of T2 are performed first, but operations of T1 are not in order (order should be R1(A), W1(A), R1(B), W1(B)). So S2 is not conflict serializable.

**Conflict Equivalent:** Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations. In the example discussed above, S11 is conflict equivalent to S1 (S1 can be converted to S11 by swapping non-conflicting operations). Similarly, S11 is conflict equivalent to S12 and so on.

***Note 1:****Although S2 is not conflict serializable, but still it is conflict equivalent to S21 and S21 because S2 can be converted to S21 and S22 by swapping non-conflicting operations.*

***Note 2:****The schedule which is conflict serializable is always conflict equivalent to one of the serial schedule. S1 schedule discussed above (which is conflict serializable) is equivalent to serial schedule (T1->T2).*

**Question:** **Consider the following schedules involving two transactions. Which one of the following statement is true?**

S1: R1(X) R1(Y) R2(X) R2(Y) W2(Y) W1(X)

S2: R1(X) R2(X) R2(Y) W2(Y) R1(Y) W1(X)

* Both S1 and S2 are conflict serializable
* Only S1 is conflict serializable
* Only S2 is conflict serializable
* None

**Solution:** Two transactions of given schedules are:

T1: R1(X) R1(Y) W1(X)

T2: R2(X) R2(Y) W2(Y)

Let us first check serializability of S1:

S1: R1(X) R1(Y) R2(X) R2(Y) W2(Y) W1(X)

To convert it to a serial schedule, we have to swap non-conflicting operations so that S1 becomes equivalent to serial schedule T1->T2 or T2->T1. In this case, to convert it to a serial schedule, we must have to swap R2(X) and W1(X) but they are conflicting. So S1 can’t be converted to a serial schedule.

Now, let us check serializability of S2:

S2: **R1(X) R2(X)** R2(Y) W2(Y) R1(Y) W1(X)

Swapping non conflicting operations R1(X) and R2(X) of S2, we get

S2’: R2(X) **R1(X) R2(Y)** W2(Y) R1(Y) W1(X)

Again, swapping non conflicting operations R1(X) and R2(Y) of S2’, we get

S2’’: R2(X) R2(Y) **R1(X) W2(Y)** R1(Y) W1(X)

Again, swapping non conflicting operations R1(X) and W2(Y) of S2’’, we get

S2’’’: R2(X) R2(Y) W2(Y) R1(X) R1(Y) W1(X)

which is equivalent to a serial schedule T2->T1.

So, **correct option is C**. Only S2 is conflict serializable.

**Precedence Graph** or **Serialization Graph** is used commonly to test Conflict Serializability of a schedule.

If the graph is acyclic, the schedule is conflict serializable.

If the graph is cyclic, we can conclude that it is **not conflict serializable** to any schedule serial schedule.

### **Recursive Relationship**

A relationship between two entities of similar entity type is called a **recursive** relationship. Here the same entity type participates more than once in a relationship type with a different role for each instance. In other words, a relationship has always been between occurrences in two different entities. However, it is possible for the same entity to participate in the relationship. This is termed a **recursive** relationship.

## **What is SQL Server Profiler?**

Microsoft SQL Server Profiler is a graphical user interface that allows system administrators to monitor events of the database engine. SQL server profiler trace monitor every event to a file. SQL profiler can be used for real-time monitoring of data or also for future analysis of data. You can do the following things with a SQL Server Profiler -

* You can create a trace.
* You can watch the trace results when the trace runs.
* You can store the trace results in a table.
* You can find out the bugs in queries and diagnose it.
* If it is necessary, you can start, stop, pause and modify the trace results.
* Microsoft SQL Server Profiler is a graphical user interface to SQL Trace for monitoring an instance of the Database Engine or Analysis Services. You can capture and save data about each event to a file or table to analyze later.
* Use SQL Profiler to monitor only the events in which you are interested.
* If traces are becoming too large, you can filter them based on the information you want, so that only a subset of the event data is collected. Monitoring too many events adds overhead to the server and the monitoring process and can cause the trace file or trace table to grow very large, especially when the monitoring process takes place over a long period of time.

## **SQL Server Agent?**

SQL Server agent is a component of Microsoft SQL Server. It is a background tool of Microsoft SQL Server, so it runs continuously in the background as a window service. SQL Server Agent allows the database administrator to handle automated tasks and schedules jobs. It runs a window service so can start automatically when the system boots or you can start it manually.

The SQL Server agent plays a vital role in day to day tasks of SQL server administrator (DBA). It is one of the essential parts of the Microsoft's SQL server. Server agent's purpose is to implement the tasks easily with the scheduler engine which allows our jobs to run at scheduled date and time. SQL server agent store scheduled administrative tasks information using SQL server.

## **SQL Injection?**

SQL injection is a code injection technique that might destroy your database. It is one of the most common web hacking techniques. SQL injection is the placement of malicious code in SQL statements, via web page input.

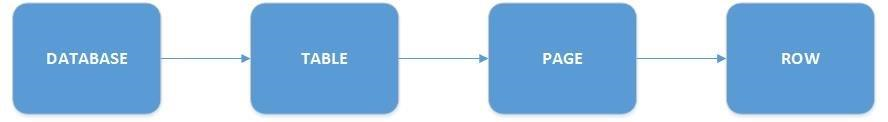
Here are five ways to protect ourselves:

1. **Use stored procedures or prepared SQL statements.** So do *not* build dynamic SQL. This is the most effective way to prevent SQL injection.
2. **Validate the type and pattern of input.** If you know you're looking for specific data—like an ID, name, or email address—validate any user input based on type, length, or other attributes.
3. **Escape special characters like quotes.** This approach is a quick and easy way to reduce the chances of SQL injection, but it's not fully effective.
4. **Limit database privileges.** Application accounts that connect to the database should have as few privileges as possible. It's unlikely, for example, that your application will ever have to delete a table. So, don't allow it.
5. **Don't display database error messages to users.** Error messages contain information that could tell hackers a lot of information about your data. Best practice is to give generic database error messages to users, and log detailed errors where developers can access them. Even better, send an *alert* to the dev team when there’s an error.

## **LOCKS**

**Lock:**Lock is a mechanism to ensure data consistency. SQL Server locks objects when the transaction starts. When the transaction is completed, SQL Server releases the locked object. This lock mode can be changed according to the SQL Server process type and isolation level.  These lock modes are:

**Lock hierarchy:**SQL Server has a lock hierarchy which acquires lock objects in this hierarchy. A database is located at the top of the hierarchy and row is located at the bottom. The below image illustrates the lock hierarchy of SQL Server.



**Shared (S) Locks:** This lock type occurs when the object needs to be read. This lock type does not cause much problem. Those locks acquired by readers during read operations such as SELECT. I’d like to mention that it happens in most part of the cases but not all the time. There are some cases when readers don’t acquire (S) locks. Shared (S) locks allow concurrent transactions to read (SELECT) a resource under pessimistic concurrency control. No other transactions can modify the data while shared (S) locks exist on the resource. Shared (S) locks on a resource are released as soon as the read operation completes, unless the transaction isolation level is set to repeatable read or higher, or a locking hint is used to retain the shared (S) locks for the duration of the transaction.

**Exclusive (X) Locks:** When this lock type occurs, it occurs to prevent other transactions to modify or access a locked object. Those locks acquired by writers during data modification operators such as Insert, Update or Delete. Those locks prevent one object to be modified by the different sessions. Those locks are always acquired and held till end of transaction. Exclusive (X) locks prevent access to a resource by concurrent transactions. With an exclusive (X) lock, no other transactions can modify data; read operations can take place only with the use of the NOLOCK hint or read uncommitted isolation level.

Data modification statements, such as INSERT, UPDATE, and DELETE combine both modification and read operations. The statement first performs read operations to acquire data before performing the required modification operations. Data modification statements, therefore, typically request both shared locks and exclusive locks. For example, an UPDATE statement might modify rows in one table based on a join with another table. In this case, the UPDATE statement requests shared locks on the rows read in the join table in addition to requesting exclusive locks on the updated rows.

**Update (U) Locks:** This lock type is similar to the exclusive lock but it has some differences. We can divide the update operation into different phases: read phase and write phase. During the read phase, SQL Server does not want other transactions to have access to this object to be changed. For this reason, SQL Server uses the update lock. Those locks are the mix between shared and exclusive locks. SQL Server uses them with data modification statements while searching for the rows need to be modified. For example, if you issue the statement like: “*update MyTable set Column1 = 0 where Column1 is null*” SQL Server acquires update lock for every row it processes while searching for*Column1 is null*. When eligible row found, SQL Server converts (U) lock to (X). Update (U) locks prevent a common form of deadlock. In a repeatable read or serializable transaction, the transaction reads data, acquiring a shared (S) lock on the resource (page or row), and then modifies the data, which requires lock conversion to an exclusive (X) lock. If two transactions acquire shared-mode locks on a resource and then attempt to update data concurrently, one transaction attempts the lock conversion to an exclusive (X) lock. The shared-mode-to-exclusive lock conversion must wait because the exclusive lock for one transaction is not compatible with the shared-mode lock of the other transaction; a lock wait occurs. The second transaction attempts to acquire an exclusive (X) lock for its update. Because both transactions are converting to exclusive (X) locks, and they are each waiting for the other transaction to release its shared-mode lock, a deadlock occurs.

To avoid this potential deadlock problem, update (U) locks are used. Only one transaction can obtain an update (U) lock to a resource at a time. If a transaction modifies a resource, the update (U) lock is converted to an exclusive (X) lock.

**Intent (IS, IX, IU) Locks:** The intent lock occurs when SQL Server wants to acquire the shared (S) lock or exclusive (X) lock on some of the resources lower in the lock hierarchy. In practice, when SQL Server acquires a lock on a page or row, the intent lock is required in the table. Those locks indicate locks on the child objects. For example, if row has (X) lock, it would introduce (IX) locks on page, table and database level. Main purpose of those locks is optimization. This about situation when you need to have exclusive access to the database (i.e. (X) lock on database level). If SQL Server did not have intent locks, it would have to scan all rows in the all objects and see if there are any low level locks acquired. The Database Engine uses intent locks to protect placing a shared (S) lock or exclusive (X) lock on a resource lower in the lock hierarchy. Intent locks are named intent locks because they are acquired before a lock at the lower level, and therefore signal intent to place locks at a lower level.

Intent locks serve two purposes:

* To prevent other transactions from modifying the higher-level resource in a way that would invalidate the lock at the lower level.
* To improve the efficiency of the Database Engine in detecting lock conflicts at the higher level of granularity.

In addition to the row level locks, SQL Server also acquires so-called **Intent Locks** at higher levels within the Lock Hierarchy: at the page and at the table level. Intent Locks have to do with performance optimization. SQL Server acquires the following Intent-Locks based on the requested row level lock:

* Intent Shared Lock (IS), when you have a Shared Lock at the row level
* Intent Update Lock (IU), when you have an Update Lock at the row level
* Intent Exclusive Lock (IX), when you have an Exclusive Lock at the row level

After all these brief explanations, we will try to find an answer to how to identify locks.  SQL Server offers a lot of dynamic management views to access metrics. To identify SQL Server locks, we can use the **sys.dm\_tran\_locks** view. In this view, we can find a lot of information about currently active lock manager resources.

1. (S) locks are compatible with (S) and (U) locks.
2. (X) locks are incompatible with any other lock types
3. (U) locks are compatible with (S) but incompatible with (U)

**Schema Locks :**

The Database Engine uses schema modification (Sch-M) locks during a table data definition language (DDL) operation, such as adding a column or dropping a table. During the time that it is held, the Sch-M lock prevents concurrent access to the table. This means the Sch-M lock blocks all outside operations until the lock is released. Some data manipulation language (DML) operations, such as table truncation, use Sch-M locks to prevent access to affected tables by concurrent operations.

The Database Engine uses schema stability (Sch-S) locks when compiling and executing queries. Sch-S locks do not block any transactional locks, including exclusive (X) locks. Therefore, other transactions, including those with X locks on a table, continue to run while a query is being compiled. However, concurrent DDL operations, and concurrent DML operations that acquire Sch-M locks, cannot be performed on the table.

**Bulk update Locks :**

Bulk update (BU) locks allow multiple threads to bulk load data concurrently into the same table while preventing other processes that are not bulk loading data from accessing the table. The Database Engine uses bulk update (BU) locks when both of the following conditions are true.

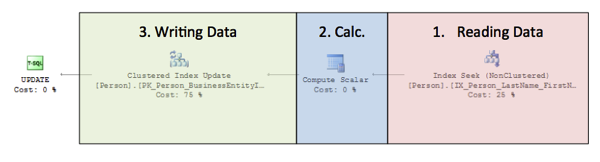
* You use the Transact-SQL BULK INSERT statement, or the OPENROWSET(BULK) function, or you use one of the Bulk Insert API commands such as .NET SqlBulkCopy, OLEDB Fast Load APIs, or the ODBC Bulk Copy APIs to bulk copy data into a table.
* The **TABLOCK** hint is specified or the **table lock on bulk load** table option is set using **sp\_tableoption**.

| **Lock mode** | **Description** |
| --- | --- |
| Shared (S) | Used for read operations that do not change or update data, such as a SELECT statement. |
| Update (U) | Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later. |
| Exclusive (X) | Used for data-modification operations, such as INSERT, UPDATE, or DELETE. Ensures that multiple updates cannot be made to the same resource at the same time. |
| Intent | Used to establish a lock hierarchy. The types of intent locks are: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX). |
| Schema | Used when an operation dependent on the schema of a table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S). |
| Bulk Update (BU) | Used when bulk copying data into a table and the **TABLOCK** hint is specified. |
| Key-range | Protects the range of rows read by a query when using the serializable transaction isolation level. Ensures that other transactions cannot insert rows that would qualify for the queries of the serializable transaction if the queries were run again. |

## **Why Update Lock is needed in SQL Server?**

In general, an Update Lock is used in SQL Server when performing an UPDATE statement. When you look at the underlying query plan, you can see that such a plan always consists of 3 parts:

* Reading data
* Calculating new values
* Writing data



When SQL Server initially reads the data to be changed in the first part of the query plan, Update Locks are acquired on the individual records. And finally, these Update Locks are converted to Exclusive (X) Locks when the data is changed in the third part of the query plan. The question that arises with this approach is always the same: why does SQL Server acquire Update Locks instead of Shared (S) Locks in the first phase? When you normally read data through a SELECT statement, a Shared Lock is also good enough. Why is there now a different approach with UPDATE query plans? Let’s have a more detailed look at it.

An Update Lock is only compatible with a Shared Lock, but isn’t compatible with another Update or Exclusive Lock. Therefore a deadlock situation can be avoided, because 2 UPDATE query plans can’t run concurrently at the same time. The 2nd query will just wait until the Update Lock can be acquired in the 1st phase of the query plan.

**Update lock is needed for Deadlock Avoidance.**

First of all UPDATE Locks are needed to avoid deadlock situations in UPDATE query plans. Let’s try to imagine what happens when multiple UPDATE query plans acquire Shared Locks in the first phase of the plan, and afterwards convert these Shared Locks to Exclusive Locks when the data is finally changed in the third phase of the query plan:

* The 1st query can’t convert the Shared Lock to an Exclusive Lock, because the 2nd query has already acquired a Shared Lock.
* The 2nd query can’t convert the Shared Lock to an Exclusive Lock, because the 1st query has already acquired a Shared Lock.

## **Snapshots**

A database snapshot is a read-only, static view of a SQL Server database (the *source database*). The database snapshot is transactionally consistent with the source database as of the moment of the snapshot's creation. A database snapshot always resides on the same server instance as its source database. As the source database is updated, the database snapshot is updated. Therefore, the longer a database snapshot exists, the more likely it is to use up its available disk space.

Multiple snapshots can exist on a given source database. Each database snapshot persists until it is explicitly dropped by the database owner.

Snapshots are the copies of read only data that is stored in the master table. Snapshots are located on remote node and refreshed periodically so that the changes in master table can be recorded. They are also replica of tables.

* Snapshot refers to a complete visualization of data at the time of extraction. It occupies less space and can be used to back up and restore data quickly.
* A snapshot is a process of knowing about the activities performed. It is stored in a report format from a specific catalog. The report is generated soon after the catalog is disconnected.

There are various scenarios where database snapshots can be useful. Some of them are listed below.

* If we want to restore our database rather than SQL backup file, we can go with database snapshot. It’s a quick way to restore a database.
* If any user wants to do any kind of manipulation with the data on temporary basis, we can create a database snapshot of the database and let user to use this snapshot instead of the original database. This seems beneficial because the user can perform their manipulation without making any changes in original database. Once done with the activity, user can delete the snapshot.
* Database Snapshot can be used for reporting purpose.
* Maintaining historical data for report generation.

The following are some important points related to database snapshot,

* DS can be created in all recovery models.
* It’s not possible to take backup of the database snapshot.
* You cannot restore, detach a DS.
* Full textindexes are not supported in DS.
* We can create multiple snapshots for a single database.
* DS can only be created by T-SQL. It’s not possible to create DS with GUI.
* If DS is created for any database, SQL Server won’t allow you to drop the source database until you drop the related DS. If you try, you’ll get error.

## **Operational Data Source (ODS)**

* An operational data store (“ODS”) is a database designed to integrate data from multiple sources for additional operations on the data. Unlike a master data store, the data is not sent back to operational systems. It may be passed for further operations and to the data warehouse for reporting.
* In ODS, data can be scrubbed, resolved for redundancy and checked for compliance with the corresponding business rules. This data store can be used for integrating disparate data from multiple sources so that business operations, analysis and reporting can be carried while business operations occur. This is the place where most of the data used in current operation is housed before it’s transferred to the data warehouse for longer term storage or archiving.
* An ODS is designed for relatively simple queries on small amounts of data (such as finding the status of a customer order), rather than the complex queries on large amounts of data typical of the data warehouse.
* An ODS is similar to your short term memory where it only stores very recent information. On the contrary, the data warehouse is more like long term memory storing relatively permanent information.

## **Different type of replication in SQL Server**

There are three types of replication in SQL Server.

**1) Snapshot replication**: Snapshot replication distributes data exactly as it appears at a specific moment. Snapshot replication is the best method for replicating data that changes infrequently. Snapshot replication is the easiest way to maintain.

**2) Transactional replication**: Transactional replication is a process of distributing data from publisher to subscriber. Transactional replication is generally used in the "server to server" environment. It is appropriate when you want incremental change propagated to the subscriber.

**3) Merge replication**: Merge replication grouped the data from various sources to a single centralized database. It is generally used in the server to the client environment. Merge replication is appropriate when multiple subscribers might update the same data at the various time.

## **Magic Tables in SQL server**

A Table which is automatically created and managed by SQL server internally to store the inserted, updated values for any DML (SELECT, DELETE, UPDATE, etc.) operation, is called as Magic tables in SQL server. The triggers preferably use it.

## **Different steps to secure SQL Server**

* Preferring NT authentication
* Using server, database and application roles to control access to the data
* Securing the [physical database files using NTFS permissions](https://intellipaat.com/tutorial/oracle-dba-tutorial/oracle-installation/)
* Using an unusable SA password, restricting physical access to the SQL Server
* Renaming the Administrator account on the SQL Server computer
* Disabling the Guest account, enabling auditing using multiprotocol encryption,
* Setting up SSL, setting up firewalls, isolating SQL Server from the web server

## **Hotfixes and Patches?**

Hotfixes are software patches that were applied to live systems (the ones still running.) A **hotfix** is a single, cumulative package that includes one or more files used to address a problem in a software product (i.e. a software bug).

A patch is a program installed in the machine to rectify the problem occurred in the system and ensured the security of that system. The hotfix is a Kind of Patches provided by the Microsoft.

In [Microsoft SQL SERVER context](https://intellipaat.com/interview-question/sql-interview-questions/), hotfixes are small patches designed to address specific issues, most commonly to freshly-discovered security holes.

Ex: If a select query returns duplicate rows with aggregations, the result may be wrong.

## **Lock Escalation?**

Lock escalation is used to convert row locks and page locks into table locks thereby “escalating” the smaller or finer locks. This increases the system performance as each lock is nothing but a memory structure. Too many locks would mean more consumption of memory. Hence, escalation is used. Lock escalation from SQL Server 7.0 onwards is dynamically managed by SQL Server. It is the process of converting a lot of low level locks into higher level locks.

## **Virtual Data Warehouse**

A virtual data warehouse provides a view of completed data. Within Virtual data warehousing, it doesn’t have any historical data and it can be considered as a logical data model which has the metadata. A virtual data warehouse is a perfect information system where it acts as an appropriate analytical decision-making system.

It is one of the best ways of portraying raw data in the form of meaningful data for executive users which makes business sense and at the same time it provides suggestions at the time of decision making.

## **Cluster Analysis**

A cluster analysis is defined as a process where an object is defined without giving any label to it. It uses statistical data analysis technique and processes the data mining job. Using cluster analysis, an iterative process of knowledge discovery is processed in the form of trails.

The purpose of cluster analysis:

1. It is scalable
2. It can deal with different set of attributes
3. High dimensionality
4. Interpretability

## **Conflict serializable and Conflict Equivalent**

[**Conflict serializable and Conflict Equivalent**](https://www.geeksforgeeks.org/conflict-serializability/): A schedule is conflict serializable if it is conflict equivalent to a serial schedule.

**Checking for Conflict Serializability**

To check whether a schedule is conflict serializable or not, find all **conflicting operations pair**s of a schedule and draw precedence graph (For all conflicting operation pair, an edge from Ti to Tj if one operation of conflicting pair is from Ti and other from Tj and operation of Ti occurs before Tj in schedule). If graph does not contain cycle, the schedule is conflict serializable else it is not conflict serializable.

Schedules are said to be conflict equivalent if 1 schedule can be converted into another by swapping non-conflicting operations.

**Note:** Two phase locking protocol produce conflict serializable schedule but may suffer from deadlock. On the other hand, Time-Stamp based protocols are free from deadlock yet produce conflict serializable schedule.

## **View serializable and View Equivalence**

[**View Serializable and View Equivalence**](https://www.geeksforgeeks.org/dbms-how-to-test-two-schedule-are-view-equal-or-not-2/) : Two schedules S1 and S2 are said to be view-equivalent if all conditions are satisfied for all objects:

* If the transaction Ti in S1 reads an initial value for object X, in S2 also, Ti must read the initial value of X.
* If the transaction Ti in S1 reads the value written by transaction Tj in S1 for object X, same should be done in S2.
* If the transaction Ti in S1 is the final transaction to write the value for an object X, in S2 also,Ti must write the final value of X.

A schedule is view serializable if it is view equivalent to any serial schedule.

## **Partitioning**

Partitioning is the database process where very large tables are divided into multiple smaller parts. By splitting a large table into smaller, individual tables, queries that access only a fraction of the data can run faster because there is less data to scan. The main goal of partitioning is to aid in **maintenance of large tables and to reduce the overall response time** to read and load data for particular SQL operations.

Table **partitioning** is a way to divide a large table into smaller, more manageable parts without having to create separate tables for each part. Data in a **partitioned** table is physically stored in groups of rows called **partitions** and each **partition** can be accessed and maintained separately.

All partitions of a single index or table must reside in the same database. The table or index is treated as a single logical entity when queries or updates are performed on the data.

There are two different approaches we could use to accomplish table partitioning. The first is to create a new partitioned table and then simply copy the data from your existing table into the new table and do a table rename. The second approach is to partition an existing table by rebuilding or creating a clustered index on the table. Partitioning can only occur in a single column and that your queries must include that column otherwise, a full scan will be performed.  For this, it is important to review the way in which your queries access the table in order to choose the most suitable column for partitioning.

**Scenario:**

“A company that maintains a large database, stores all of the data that are produced as a result of its activity. As time goes by, the velocity of the data increases more and more and queries become slower and slower as whole tables need to be scanned. But what happens in cases where there is no need for a full scan? Imagine the compilation of monthly business intelligence reports. The only data that are actually needed are those that were produced during the last month. It becomes evident that there are cases where it would be extremely helpful to be able to have control over the data that our queries take into consideration while being evaluated. ”

**Benefits of Partitioning:**

Partitioning large tables or indexes can have the following manageability and performance benefits.

* You can transfer or access subsets of data quickly and efficiently, while maintaining the integrity of a data collection. For example, an operation such as loading data from an OLTP to an OLAP system takes only seconds, instead of the minutes and hours the operation takes when the data is not partitioned.
* You can perform maintenance operations on one or more partitions more quickly. The operations are more efficient because they target only these data subsets, instead of the whole table. For example, you can choose to compress data in one or more partitions or rebuild one or more partitions of an index.
* You may improve query performance, based on the types of queries you frequently run and on your hardware configuration. For example, the query optimizer can process equi-join queries between two or more partitioned tables faster when the partitioning columns are the same as the columns on which the tables are joined.
* The relative speedup of queries that require only portions of large data sets. In this case, the optimizer eliminates searching in partitions that do not have relevant information.
* Faster data load
* Faster deletion of old data limited to certain partitions, if they are no longer needed.
* Faster archival of rarely used or old data can be migrated to cheaper and slower storage media.

**Creating a partitioned table or index typically happens in four parts:**

1. Create a filegroup or filegroups and corresponding files that will hold the partitions specified by the partition scheme.
2. Create a partition function that maps the rows of a table or index into partitions based on the values of a specified column.
3. Create a partition scheme that maps the partitions of a partitioned table or index to the new filegroups.
4. Create or modify a table or index and specify the partition scheme as the storage location.

**Limitations and Restrictions:**

* The scope of a partition function and scheme is limited to the database in which they have been created. Within the database, partition functions reside in a separate namespace from other functions.
* If any rows within a partition function have partitioning columns with null values, these rows are allocated to the left-most partition. However, if NULL is specified as a boundary value and RIGHT is indicated, the left-most partition remains empty and NULL values are placed in the second partition.

**Permissions:**

Creating a partitioned table requires CREATE TABLE permission in the database and ALTER permission on the schema in which the table is being created. Creating a partitioned index requires ALTER permission on the table or view where the index is being created. Creating either a partitioned table or index requires any one of the following additional permissions:

* ALTER ANY DATASPACE permission. This permission defaults to members of the **sysadmin** fixed server role and the **db\_owner** and **db\_ddladmin** fixed database roles.
* CONTROL or ALTER permission on the database in which the partition function and partition scheme are being created.
* CONTROL SERVER or ALTER ANY DATABASE permission on the server of the database in which the partition function and partition scheme are being created.

## **Vertical Partitioning**

Vertical table partitioning is mostly used to increase SQL Server performance especially in cases where a query retrieves all columns from a table that contains a number of very wide text or BLOB columns. In this case to reduce access times the BLOB columns can be split to its own table. Another example is to restrict access to sensitive data e.g. passwords, salary information etc. Vertical partitioning splits a table into two or more tables containing different columns.

For eg: An example for vertical partitioning can be a large table with reports for employees containing basic information, such as report name, id, number of report and a large column with report description. Assuming that ~95% of users are searching on the part of the report name, number, etc. and that only ~5% of requests are opening the reports description field and looking to the description. Let’s assume that all those searches will lead to the clustered index scans and since the index scan reads all rows in the table the cost of the query is proportional to the total number of rows in the table and our goal is to minimize the number of IO operations and reduce the cost of the search.

Vertical partitioning on SQL Server tables may not be the right method in every case. However, if you have, for example, a table with a lot of data that is not accessed equally, tables with data you want to restrict access to, or scans that return a lot of data, vertical partitioning can help.

## **Horizontal Partitioning**

Horizontal partitioning divides a table into multiple tables that contain the same number of columns, but fewer rows. For example, if a table contains a large number of rows that represent monthly reports it could be partitioned horizontally into tables by years, with each table representing all monthly reports for a specific year. This way queries requiring data for a specific year will only reference the appropriate table. Tables should be partitioned in a way that queries reference as few tables as possible.

Tables are horizontally partitioned based on a column which will be used for partitioning and the ranges associated to each partition. Partitioning column is usually a datetime column but all data types that are valid for use as index columns can be used as a partitioning column, except a timestamp column. The ntext, text, image, xml, varchar(max), nvarchar(max), or varbinary(max), Microsoft .NET Framework common language runtime (CLR) user-defined type, and alias data type columns cannot be specified.

There are two different approaches we could use to accomplish table partitioning. The first is to create a new partitioned table and then simply copy the data from your existing table into the new table and do a table rename. The second approach is to partition an existing table by rebuilding or creating a clustered index on the table.

SQL Server 2005 introduced a built-in partitioning feature to horizontally partition a table with up to 1000 partitions in SQL Server 2008, and 15000 partitions in SQL Server 2012, and the data placement is handled automatically by SQL Server. This feature is available only in the Enterprise Edition of SQL Server.

To create a partitioned table for storing monthly reports we will first create additional filegroups. A filegroup is a logical storage unit. Every database has a primary filegroup that contains the primary data file (.mdf). An additional, user-defined, filegrups can be created to contain secondary files (.ndf). **We will create 12 filegroups for every month:**

| ALTER DATABASE PartitioningDB  ADD FILEGROUP January  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP February  GO  ALTER DATABASE PartitioningDB  ADD FILEGROUP March  GO |  |
| --- | --- |

**To check created and available file groups in the current database run the following query:**

| SELECT name AS AvailableFilegroups  FROM sys.filegroups  WHERE type = 'FG' |
| --- |

**When filegroups are created we will add .ndf file to every filegroup:**

| ALTER DATABASE [PartitioningDB]      ADD FILE      (      NAME = [PartJan],      FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL11.LENOVO\MSSQL\DATA\PartitioningDB.ndf',          SIZE = 3072 KB,          MAXSIZE = UNLIMITED,          FILEGROWTH = 1024 KB      ) TO FILEGROUP [January] |
| --- |
|  |

**The same way files to all created filegroups with specifying the logical name of the file and the operating system (physical) file name for each filegroup e.g.:**

| ALTER DATABASE [PartitioningDB]      ADD FILE      (      NAME = [PartFeb],      FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL11.LENOVO\MSSQL\DATA\PartitioningDB2.ndf',          SIZE = 3072 KB,          MAXSIZE = UNLIMITED,          FILEGROWTH = 1024 KB      ) TO FILEGROUP [February] |
| --- |

**To check files created added to the filegroups run the following query:**

| SELECT  name as [FileName],  physical\_name as [FilePath]  FROM sys.database\_files  where type\_desc = 'ROWS'  GO |
| --- |

**After creating additional filegroups for storing data we’ll create a partition function.** A partition function is a function that maps the rows of a partitioned table into partitions based on the values of a partitioning column. In this example we will create a partitioning function that partitions a table into 12 partitions, one for each month of a year’s worth of values in a datetime column:

| CREATE PARTITION FUNCTION [PartitioningByMonth] (datetime)  AS RANGE RIGHT FOR VALUES ('20140201', '20140301', '20140401',                 '20140501', '20140601', '20140701', '20140801',                 '20140901', '20141001', '20141101', '20141201'); |
| --- |

**To map the partitions of a partitioned table to filegroups and determine the number and domain of the partitions of a partitioned table we will create a partition scheme:**

| CREATE PARTITION SCHEME PartitionBymonth  AS PARTITION PartitioningBymonth  TO (January, February, March,      April, May, June, July,      Avgust, September, October,      November, December); |
| --- |

**Now we’re going to create the table using the PartitionBymonth partition scheme, and fill it with the test data:**

| CREATE TABLE Reports  (ReportDate datetime PRIMARY KEY,  MonthlyReport varchar(max))  ON PartitionBymonth (ReportDate);  GO  INSERT INTO Reports (ReportDate,MonthlyReport)  SELECT '20140105', 'ReportJanuary' UNION ALL  SELECT '20140205', 'ReportFebryary' UNION ALL  SELECT '20140308', 'ReportMarch' UNION ALL  SELECT '20140409', 'ReportApril' UNION ALL  SELECT '20140509', 'ReportMay' UNION ALL  SELECT '20140609', 'ReportJune' UNION ALL  SELECT '20140709', 'ReportJuly' UNION ALL  SELECT '20140809', 'ReportAugust' UNION ALL  SELECT '20140909', 'ReportSeptember' UNION ALL  SELECT '20141009', 'ReportOctober' UNION ALL  SELECT '20141109', 'ReportNovember' UNION ALL  SELECT '20141209', 'ReportDecember' |
| --- |

**We will now verify the rows in the different partitions:**

| SELECT p.partition\_number AS PartitionNumber,  f.name AS PartitionFilegroup, p.rows AS NumberOfRows  FROM sys.partitions p  JOIN sys.destination\_data\_spaces dds ON p.partition\_number = dds.destination\_id  JOIN sys.filegroups f ON dds.data\_space\_id = f.data\_space\_id  WHERE OBJECT\_NAME(OBJECT\_ID) = 'Reports' |
| --- |

**Now just copy data from your table and rename a partitioned table.**

## **Terms related to Partitioning**

**Partition Function:**

A database object that defines how the rows of a table or index are mapped to a set of partitions based on the values of certain column, called a partitioning column. That is, the partition function defines the number of partitions that the table will have and how the boundaries of the partitions are defined. For example, given a table that contains sales order data, you may want to partition the table into twelve (monthly) partitions based on a **datetime** column such as a sales date.

**Partition Scheme:**

A database object that maps the partitions of a partition function to a set of filegroups. The primary reason for placing your partitions on separate filegroups is to make sure that you can independently perform backup operations on partitions. This is because you can perform backups on individual filegroups.

**Partitioning Column:**

The column of a table or index that a partition function uses to partition the table or index. Computed columns that participate in a partition function must be explicitly marked PERSISTED. All data types that are valid for use as index columns can be used as a partitioning column, except **timestamp**. The **ntext**, **text**, **image**, **xml**, **varchar(max)**, **nvarchar(max)**, or **varbinary(max)** data types cannot be specified. Also, Microsoft .NET Framework common language runtime (CLR) user-defined type and alias data type columns cannot be specified.

**Aligned Index:**

An index that is built on the same partition scheme as its corresponding table. When a table and its indexes are in alignment, SQL Server can switch partitions quickly and efficiently while maintaining the partition structure of both the table and its indexes. An index does not have to participate in the same named partition function to be aligned with its base table. However, the partition function of the index and the base table must be essentially the same, in that:

1. The arguments of the partition functions have the same data type.
2. They define the same number of partitions.
3. They define the same boundary values for partitions.

**Partition Clustered Index:**

When partitioning a clustered index, the clustering key must contain the partitioning column. When partitioning a nonunique clustered index, and the partitioning column is not explicitly specified in the clustering key, SQL Server adds the partitioning column by default to the list of clustered index keys. If the clustered index is unique, you must explicitly specify that the clustered index key contain the partitioning column.

**Partition Non-Clustered Index:**

When partitioning a unique non-clustered index, the index key must contain the partitioning column. When partitioning a non-unique, non-clustered index, SQL Server adds the partitioning column by default as a non-key (included) column of the index to make sure the index is aligned with the base table. SQL Server does not add the partitioning column to the index if it is already present in the index.

**Non-Aligned Index:**

An index partitioned independently from its corresponding table. That is, the index has a different partition scheme or is placed on a separate filegroup from the base table. Designing an non-aligned partitioned index can be useful in the following cases:

* The base table has not been partitioned.
* The index key is unique and it does not contain the partitioning column of the table.
* You want the base table to participate in collocated joins with more tables using different join columns.

**Partition Elimination:**

The process by which the query optimizer accesses only the relevant partitions to satisfy the filter criteria of the query.

## **Query Hints**

* **NOLOCK**: In the event that data is locked, this tells SQL Server to read data from the last known value available, also known as a dirty read. Since it is possible to use some old values and some new values, data sets can contain inconsistencies. Do not use this in any place in which data quality is important.
* **RECOMPILE**: Adding this to the end of a query will result in a new execution plan being generated each time this query executed. This should not be used on a query that is executed often, as the cost to optimize a query is not trivial. For infrequent reports or processes, though, this can be an effective way to avoid undesired plan reuse. This is often used as a bandage when statistics are out of date or parameter sniffing is occurring.
* **MERGE/HASH/LOOP**: This tells the query optimizer to use a specific type of join as part of a join operation. This is super-risky as the optimal join will change as data, schema, and parameters evolve over time. While this may fix a problem right now, it will introduce an element of technical debt that will remain for as long as the hint does.
* **OPTIMIZE FOR**: Can specify a parameter value to optimize the query for. This is often used when we want performance to be controlled for a very common use case so that outliers do not pollute the plan cache. Similar to join hints, this is fragile and when business logic changes, this hint usage may become obsolete.

## **Query Processing**

**Query Processing** includes translations on high level Queries into low level expressions that can be used at physical level of file system, query optimization and actual execution of query to get the actual result.

It is done in the following steps:

* **Step-1:**  
  **Parser:** During parse call, the database performs the following checks- Syntax check, Semantic check and Shared pool check, after converting the query into relational algebra.

Parser performs the following checks as (refer detailed diagram):

* 1. **Syntax check –** concludes SQL syntactic validity. Example: Here error of wrong spelling of FROM is given by this check.
  2. **Semantic check –** determines whether the statement is meaningful or not. Example: query contains a tablename which does not exist is checked by this check.
  3. **Shared Pool check –** Every query possess a hash code during its execution. So, this check determines existence of written hash code in shared pool if code exists in shared pool then database will not take additional steps for optimization and execution.

**Hard Parse and Soft Parse –**  
If there is a fresh query and its hash code does not exist in shared pool then that query has to pass through from the additional steps known as hard parsing otherwise if hash code exists then query does not passes through additional steps. It just passes directly to execution engine (refer detailed diagram). This is known as soft parsing.  
Hard Parse includes following steps – Optimizer and Row source generation.

* **Step-2:**  
  **Optimizer:** During optimization stage, database must perform a hard parse atleast for one unique DML statement and perform optimization during this parse. This database never optimizes DDL unless it includes a DML component such as subquery that require optimization.

It is a process in which multiple query execution plan for satisfying a query are examined and most efficient query plan is satisfied for execution.  
Database catalog stores the execution plans and then optimizer passes the lowest cost plan for execution.

**Row Source Generation –**  
The Row Source Generation is a software that receives a optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database. the iterative plan is the binary program that when executes by the sql engine produces the result set.

* **Step-3:**  
  **Execution Engine:** Finally runs the query and display the required result.

## **Query optimization**

The term query optimization specifies an efficient execution plan for evaluating a query that has the least estimated cost. The concept of query optimization came into the frame when there were a number of methods, and algorithms existed for the same task then the question arose that which one is more efficient and the process of determining the efficient way is known as query optimization.

Fixing bad queries and resolving performance problems can involve hours (or days) of research and testing. Sometimes we can quickly cut that time by identifying common design patterns that are indicative of poorly performing TSQL.

There are many benefits of query optimization:

* It reduces the time and space complexity.
* More queries can be performed as due to optimization every query comparatively takes less time.
* User satisfaction as it will provide output fast

## **Query optimization techniques**

There are always 2 things to consider when optimising queries:

* What indexes can be used (you may need to create indexes)
* How the query is written (you may need to change the query to allow the query optimser to be able to find appropriate indexes, and to not re-read data redundantly)

**A few observations:**

1. You shouldn't need the subqueries - just do the direct joins and aggregate
2. You should be able to use INNER JOINs, which are typically more performant than OUTER JOINs.
3. SELECT fields instead of using SELECT \*
4. Avoid SELECT DISTINCT: To avoid using SELECT DISTINCT, select more fields to create unique results.
5. Use WHERE instead of HAVING to define filters: Per the [SQL Order of Operations](https://www.sisense.com/blog/sql-query-order-of-operations/), HAVING statements are calculated after WHERE statements. If the intent is to filter a query based on conditions, a WHERE statement is more efficient.
6. To check the existence of records, use EXISTS() rather than COUNT(): he EXISTS() method is more effective as it exits processing as soon as it finds the first entry of the record in the table. The COUNT() method would scan the entire table to return the number of records in the table that match the provided constraint.
7. You are performing date manipulations before you join your dates. As a general rule this will prevent a query optimser from using an index even if it exists. You should try to write your expressions in such a way that indexed columns exist unaltered on one side of the expression.
8. Your subqueries are filtering to the same date range as generate\_series. This is a duplication, and it limits the optimser's ability to choose the most efficient optimisation. I suspect that may have been written in to improve performance because the optimser was unable to use an index on the date column (body\_time)?
9. NOTE: We would actually very much like to use an index on Body.body\_time
10. ORDER BY within the subqueries is at best redundant. At worst it could force the query optimiser to sort the result set before joining; and that is not necessarily good for the query plan. Rather only apply ordering right at the end for final display.
11. Use of LEFT JOIN in your subqueries is inappropriate. Assuming you're using ANSI conventions for NULL behaviour (and you should be), any outer joins to envelope would return envelope\_command=NULL, and these would consequently be excluded by the condition envelope\_command=?.
12. Subqueries o and i are almost identical save for the envelope\_command value. This forces the optimser to scan the same underlying tables twice. You can use a pivot table technique to join to the data once, and split the values into 2 columns.
13. OR in the Join Predicate/WHERE Clause Across Multiple Columns:
    * SQL Server can efficiently filter a data set using indexes via the WHERE clause or any combination of filters that are separated by an AND operator. By being exclusive, these operations take data and slice it into progressively smaller pieces, until only our result set remains.
    * OR is a different story. Because it is inclusive, SQL Server cannot process it in a single operation. Instead, each component of the OR must be evaluated independently. When this expensive operation is completed, the results can then be concatenated and returned normally.
    * SQL Server cannot easily process an OR condition across multiple columns. The best way to deal with an OR is to eliminate it (if possible) or break it into smaller queries. Breaking a short and simple query into a longer, more drawn-out query may not seem elegant, but when dealing with OR problems, it is often the best choice. (eg: using UNION to divide the joins into two)
14. Wildcard String Searches:
    * String searching efficiently can be challenging, and there are far more ways to grind through strings inefficiently than efficiently. For frequently searched string columns, we need to ensure that: Indexes are present on searched columns. Those indexes can be used. If not, can we use full-text indexes? If not, can we use hashes, n-grams, or some other solution?
15. Large write operations: Large write operations are the poster-child for contention as they will often lock an entire table during the time it takes to update the data, check constraints, update indexes, and process triggers (if any exist).
16. Missing Index: Always check for indexes
17. Table size: If your query hits one or more tables with millions of rows or more, it could affect performance. Filtering the data to include only the observations you need can dramatically improve query speed. How you do this will depend entirely on the problem you're trying to solve. For example, if you've got time series data, limiting to a small time window can make your queries run much more quickly
18. Joins: If your query joins two tables in a way that substantially increases the row count of the result set, your query is likely to be slow. There's an example of this in the [subqueries lesson](https://mode.com/sql-tutorial/sql-sub-queries#joining-subqueries).
19. Aggregations: Combining multiple rows to produce a result requires more computation than simply retrieving those rows.
20. EXPLAIN: You can add EXPLAIN at the beginning of any (working) query to get a sense of how long it will take. It's not perfectly accurate, but it's a useful tool.

## **Factors that affect Query Performance/Optimization?**

The term query optimization specifies an efficient execution plan for evaluating a query that has the least estimated cost. The concept of query optimization came into the frame when there were a number of methods, and algorithms existed for the same task then the question arose that which one is more efficient and the process of determining the efficient way is known as query optimization.

There are many benefits of query optimization:

* It reduces the time and space complexity.
* More queries can be performed as due to optimization every query comparatively takes less time.
* User satisfaction as it will provide output fast

When retrieving data from SQL Server tables, don’t retrieve more than you need. Obtaining excessive data is resource expensive and time consuming.

Query performance also depends on data volume and transaction concurrency. Executing the same query on a table with millions of records requires more time that performing the same operation on the same table with only thousands of records.

A lot of concurrent transactions can degrade SQL Server performance. The shorter the queue of transactions that wait to be processed, the better performance. Executing a SQL query to retrieve records from multiple joined tables with small sets of data in a sandbox is quick, but running the same query in production with millions of rows in each joined table and multiple users accessing the same tables and data can add significant pressure. That’s why sometimes developers are not fully aware of query performance.

To be able to see how a SQL query performs in real production environment, it’s necessary to provide the same conditions as in the production environment. Otherwise, the potential underperforming can be masked.

We’ll use STATISTICS TIME to show the number of milliseconds required to parse, compile, and execute a SELECT statement.

SET STATISTICS TIME ON

SET STATISTICS TIME OFF

STATISTICT TIME shows two execution times. The first one is the CPU time, it presents the number of milliseconds the processor has been processing the data. The second time named ‘elapsed time’ is the time needed to show the results in the grid.

## **Data Segmentation**

Segmentation is organizing your database by grouping together contacts and companies based on specific characteristics. Everyone at your company can leverage segmentation. Data segmentation is vital for companies looking to improve sales and marketing productivity as it will help you improve your lead generation efforts, as well as help you gain key insights into existing customers.

Data Segmentation is the process of taking the data you hold and dividing it up and grouping similar data together based on the chosen parameters so that you can use it more efficiently within marketing and operations. Examples of Data Segmentation could be:

* Gender
* Customers vs. Prospects
* Industry

Data segmentation is how you divide and organize your data into defined groups, so you can sort through it and view it more easily. Segmented data will provide your team with clear, actionable information that can be used in your sales and marketing.

We’ll focus on data segmentation for sales and marketing (sometimes referred to as sales segmentation). But, data segmentation isn’t only useful for sales and marketing.

We’ll also look at how you can apply data segmentation to your existing customer database to identify insights that can improve customer satisfaction, and grow revenue.

Marketing can use it to determine outreach and to tailor communications. Your sales team can use it to prioritize the best-fit leads. Your service team can use segments to understand where they need to focus most and how they can provide the most relevant resources to customers.

At a higher level, segmentation can help you identify where to invest, maintain and evaluate in your customer base: Which segments are lower-performing but have high potential? Which segments are doing well? Which segments are doing poorly?

Once you know what segments are good and bad, you can then look at where they came from to determine how you can optimize your marketing and sales efforts to acquire more high-performing segments and less poor-performing ones.

Segmentation occurs in your [CRM](https://www.newbreedrevenue.com/blog/what-is-crm-software), where your contacts are stored. In order to segment your database, certain data points need to be captured and associated with individual contacts.

**The key benefits of Data Segmentation are:**

* You will be able create messaging that is tailored and sophisticated to suit your target market – appealing to their needs better.
* It allows you to easier conduct an [analysis](https://www.experian.co.uk/business/data-management/data-profiling/) of your data stored in your database, helping to identify potential opportunities and challenges based within it.
* Enables you to mass-personalise your marketing communications, reducing costs.

**Five reasons to segment the database:**

While a seemingly small detail in the grand scheme of things, segmentation has the ability to derail your entire [inbound marketing strategy](https://www.revenueriver.co/digital-growth) when not done properly or not done at all. This little step can lay the foundation for a solid marketing plan, dictate the way your content is perceived, the quality of your database, and more.

While it may require a little more due diligence, the payoff will be worth it.

**1) Segmentation Gets Your Content in the Right Hands**

What’s the point of slaving away on your content marketing efforts if the messages within your content don’t apply to the people receiving it and viewing it? There is no point, but sadly, it happens more often than not, unless of course you’ve taken us up on our advice and segmented with a purpose in mind. By thoughtfully segmenting your database, [you can tailor the message of your emails based on what you know the subgroups of your contacts want to hear about](https://blog.hubspot.com/blog/tabid/6307/bid/32848/why-list-segmentation-matters-in-email-marketing.aspx?__hstc=230351747.55edbe6317d69803e9576e4b12ae3454.1621402460449.1621402460449.1621402460449.1&__hssc=230351747.1.1621402460450&__hsfp=322626364). For instance, if John Smith fills out a form and denotes that he is interested in Product X, but is not interested in Product Y, start sending him emails about Product X and STOP sending him emails about Product Y.

**2) Segmentation Leads to Effective Automation**

While HubSpot’s automation features go above and beyond, [your automation strategy still requires a certain level of thought and forward thinking](https://blog.hubspot.com/customers/how-to-segment-and-nurture-your-customers-with-hubspot?__hstc=230351747.55edbe6317d69803e9576e4b12ae3454.1621402460449.1621402460449.1621402460449.1&__hssc=230351747.1.1621402460450&__hsfp=322626364). Therefore, stop automating just to automate, and start doing so with a purpose. This purpose should begin with segmentation. When you intertwine segmentation into your automation strategy, your automation results will improve significantly. Rather than sending broad, generalized content via automation to your contacts, start sending targeted messages to a targeted group. For instance, people who are not in a decision-making role should not be sent emails that are geared towards people who can make purchase decisions for their company. In another example, the people who do not open an email with top-of-the-funnel messaging and content, should not receive the next email in that series, which offers middle-of-the-funnel content.

**3) Segmentation Allows You to Gauge Your Database**

What’s the point of having thousands of leads in your database if you know nothing about them and don’t know how to connect with them? Without segmentation, you have no insight into who makes up your database and what your contacts are looking for.

By properly segmenting, you will be able to determine the type of people who have shown interest in your specific services or offerings, the issues or questions contacts are running into, and demographic information. With this information and data being known, you will be able to determine what you need to be doing from an inbound marketing perspective, as well as the traits and interests of your audience. For instance, if you discover, by means of your segmentation strategy, that a large portion of your database is made up of women in their mid-forties, stop sending that portion of your database information about products geared towards men.

**4) Segmentation Keeps Your Database Clean**

In addition to creating segments that are relevant to your business and are important to nurture, also keep track of and segment the irrelevant leads that you don’t see a future with. As backwards as it may sound, you need to put unqualified contacts into a bucket so they do not get mixed in with the good leads. By doing so, you are ensuring you are not wasting your time or resources on these leads. For example, if your service area is strictly within the United States and you have no future plans to expand outside of the United States, but a number of contacts within your database reside in different countries, you don’t want to waste your time on these contacts. Instead, keep track of who these contacts are.

The only thing that these contacts are doing is getting you one step closer to tipping your HubSpot contact limit. Therefore, create a HubSpot smart list with criteria that add people based on the contact property “Country” not equaling the United States. Also, be sure to keep track of and monitor [the contacts in your database who have unsubscribed from your emails and the contacts who do not engage with your emails](https://blog.hubspot.com/customers/tips-success-segmentation?__hstc=230351747.55edbe6317d69803e9576e4b12ae3454.1621402460449.1621402460449.1621402460449.1&__hssc=230351747.1.1621402460450&__hsfp=322626364). Keep your database relevant and clean by setting a reminder to go through and delete these contacts every few weeks.

**5) Segmentation Reveals Areas for Improvement**

In addition to gauging your database, segmentation can take your strategy to the next level by setting the foundation for organization-wide improvements. Adding this step to your marketing plan can allow you to see who truly makes up your database. Demographic information, interests, and other traits across your database can be useful even beyond the marketing department. Your branding, messaging, and offerings can be modified to better suit the majority of leads in your CRM. Additionally, finding new ways to segment can identify a new audience or a new opportunity for marketing activity.

**How to segment your database:**

To begin, think back to your marketing goals. Who constitutes your target market? Thinking in terms of the personas that make up your target market is a good place to start. Taking it a step further, how can you identify and group individuals into those personas? Once you have identified a way to bucket contacts into HubSpot’s persona smart lists, think collecting information like job title on a web form, then you can work towards further segmenting your database. In what other ways would you like to identify and group the hundreds or thousands of individuals that make up your database? What information would be helpful for sales to know when you hand over a lead? What information would make someone a qualified lead? And what type of information or criteria would disqualify someone in the sales process?

Create contact properties that will serve to answer whether someone is the right fit for your business and worthy of your marketing efforts, input these properties on your website forms as form fields, and create smart lists that will identify segments by means of how people answer the questions on your forms. Ultimately, you will have organized groups of people that will allow you to target the right people with the right material, which brings us to our next point.

To implement the right kind of Data Segmentation and to communicate more effectively with your target group requires a blend between having the right processes and technology in place (such as [data quality tools](https://www.experian.co.uk/business/data-management/data-processing/pandora/index) and [customer data validation](https://www.experian.co.uk/business/data-management/data-validation/)). This allows you to analyse and profile your current database, whilst ensuring any incoming data is also segmented accordingly. A key requirement of Data Segmentation is high-quality data, in terms of it being both accurate and does not lack basic information such as “name” or “address”.

## **Temporal Table**

SQL Server 2016 introduced support for temporal tables (also known as system-versioned temporal tables) as a database feature that brings built-in support for providing information about data stored in the table at any point in time rather than only the data that is correct at the current moment in time. Temporal is a database feature that was introduced in ANSI SQL 2011.

Temporal tables, also known as system-versioned tables, provide us with new functionality to track data changes. It allows SQL Server to maintain and manage the history of the data in the table automatically. This feature provides a full history of every change made to the data.

Temporal tables are considered as one of the critical features used to audit SQL Server data.

The two period columns which are of datetime2 data type, **SysStartTime** and **SysEndTime** define the current and historical data for every row of the table. It defines the validity of the data. The DateTime range has a value that is appropriate during that timeframe. By querying the table, we can easily see the transitioning of the data to its various states within the specific date-time intervals.

The following are some usage scenarios of Temporal tables

1. Auditing: Auditing all data changes and performing data forensics when necessary
2. Projecting and reporting for historical trend analysis: Reconstructing state of the data as of any time in the past
3. Calculating trends over time
4. Maintaining a slowly changing dimension for decision support applications
5. Protecting the data in case of accidental data loss: Recovering from accidental data changes and application errors
6. Rebuilding the data in case of inadvertent changes

The following query searches for row versions for Employee row with EmployeeID = 1000 that were active at least for a portion of period between 1st January of 2014 and 1st January 2015:

SELECT \* FROM Employee

FOR SYSTEM\_TIME

BETWEEN '2014-01-01 00:00:00.0000000' AND '2015-01-01 00:00:00.0000000'

WHERE EmployeeID = 1000 ORDER BY ValidFrom;

## **Temp tables in SQL**

There are four main types for the SQL temp tables:

* a **Local SQL Server temp table**, which is named starting with a # symbol (e.g. #TempShipments), that can be referenced only by the current database session and discarded by its disconnection,
* a**Global SQL temp table**, which is named starting with ## (e.g. ##TempShipments), that can be referenced by any process in the current database and discarded when the original database session that created that temp table disconnected or until the last statement that was referencing the temp table has stopped using it, as anyone who has access to the system TempDB database when that global SQL temp table is created will be able to use that table,
* a **Persistent SQL Server temp table** which is named starting with a TempDB prefix such as TempDB.DBO.TempShipments
* and a **Table Variable** that starts with an @ prefix (e.g. @TempShipments)

## **Windows function**

Window functions operate on a set of rows and return a single value for each row from the underlying query. The term window describes the set of rows on which the function operates. A window function uses values from the rows in a window to calculate the returned values.

When you use a window function in a query, define the window using the OVER() clause. The OVER() clause (window definition) differentiates window functions from other analytical and reporting functions. A query can include multiple window functions with the same or different window definitions.

The OVER() clause has the following capabilities:

* Defines window partitions to form groups of rows. (PARTITION BY clause)
* Orders rows within a partition. (ORDER BY clause)

You can also include the optional PARTITION BY and ORDER BY clauses in a query. The PARTITION BY clause subdivides the window into partitions. The ORDER BY clause defines the logical order of the rows within each partition of the result set. Window functions are applied to the rows within each partition and sorted according to the order specification.

Following are the types of Windows function:

1. Value: First\_Vaue(), Lead()
2. Aggregate: Sum(), Avg(), MIN()
3. Ranking: Cume\_Dist(), Rank(), NTILE()

Points to note:

* You can only use window functions in the SELECT list and ORDER BY clauses of a query.
* Window functions precede ORDER BY.
* Drill processes window functions after the WHERE, GROUP BY, and HAVING clauses.
* Including the OVER() clause after an aggregate set function turns the function into an aggregate window function.
* You can use window functions to aggregate over any number of rows in the window frame.

**LEAD:** Accesses data from a subsequent row in the same result set without the use of a self-join starting with SQL Server 2012. LEAD provides access to a row at a given physical offset that follows the current row. Use this analytic function in a SELECT statement to compare values in the current row with values in a following row.

**LAG:** Accesses data from a previous row in the same result set without the use of a self-join starting with SQL Server 2012 (11.x). LAG provides access to a row at a given physical offset that comes before the current row. Use this analytic function in a SELECT statement to compare values in the current row with values in a previous row.

**FIRST\_VALUE:** Returns the first value in an ordered set of values in SQL Server 2019.

**LAST\_VALUE:** Returns the last value in an ordered set of values in SQL Server 2019.

## **Synonym**

In SQL Server, a synonym is an alias or alternative name for a database object such as a table, [view](http://www.sqlservertutorial.net/sql-server-views/), [stored procedure](http://www.sqlservertutorial.net/sql-server-stored-procedures/), [user-defined function](http://www.sqlservertutorial.net/sql-server-user-defined-functions/), and [sequence](http://www.sqlservertutorial.net/sql-server-basics/sql-server-sequence/).

* Create a synonym

CREATE SYNONYM suppliers

FOR [ server\_name.[ database\_name ] . [ schema\_name\_2 ]. object\_name

* List all synonym

SELECT    name,     base\_object\_name,     type

FROM     sys.synonyms

* Drop a synonym

DROP SYNONYM IF EXISTS orders;

**Uses of Synonymns:**

* Simplify object names
* Enable seamless object name changes
* Provide a layer of abstraction over the base objects.
* Shorten the lengthy name e.g., a very\_long\_database\_name.with\_schema.and\_object\_name with a simplified alias.
* Allow backward compatibility for the existing applications when you rename database objects such as tables, views, stored procedures, user-defined functions, and sequences.

## **Log Shipping**

Log shipping is the process of automating the backup of a database and transaction log file on a primary database server and then restoring them on a standby server. Many servers support this technique for maintaining a backup server, such as Microsoft SQL Server, 4D server, MySQL, and PostgreSQL.

The primary purpose of log shipping is to increase database availability just like replication. On each of secondary database, log backups are applied particularly.

Steps for log shipping process:

* Firstly take a backup of transaction log file on Primary SQL server instance
* Copy the log file on secondary SQL server instance
* Restore the Log backup file onto secondary SQL Server instance

## **Types of replication in SQL Server?**

There are three types of replication in SQL Server.

**1) Snapshot replication**: Snapshot replication distributes data exactly as it appears at a specific moment. Snapshot replication is the best method for replicating data that changes infrequently. Snapshot replication is the easiest way to maintain.

**2) Transactional replication**: Transactional replication is a process of distributing data from publisher to subscriber. Transactional replication is generally used in the "server to server" environment. It is appropriate when you want incremental change propagated to the subscriber.

**3) Merge replication**: Merge replication grouped the data from various sources to a single centralized database. It is generally used in the server to the client environment. Merge replication is appropriate when multiple subscribers might update the same data at the various time.

## **Magic Tables in SQL server**

A Table which is automatically created and managed by SQL server internally to store the inserted, updated values for any DML (SELECT, DELETE, UPDATE, etc.) operation, is called as Magic tables in SQL server. The triggers preferably use it.

## **CDC in SQL Server**

CDC is termed as "Change Data Capture." It captures the recent activity of INSERT, DELETE, and UPDATE, which are applied to the SQL Server table. It records the changes made in the SQL server table in a compatible format.

## **Collation sensitivity? Explain different types.**

Collation sensitivity is used to define the rules for sorting and comparing the strings of character data. The basic rule for sorting a character data are correct character sequence, Case-sensitivity, character width, and accent marks, etc.

**Different types of collation sensitivity:**

* Case sensitivity
* Accent sensitivity
* Kana sensitivity
* Width sensitivity

**Case Sensitivity**: Case sensitivity defines every character with a unique value, as alphabet characters A and a are treated individually, as they have different ASCII values for a computer language

**Accent sensitivity**: Accent sensitivity is related that whether the accent is off or not, as a and á both should be treated differently

**Kana sensitivity**: Kana sensitivity defines the difference between two Japanese words: Hiragana and Katakana

**Width sensitivity**: It differentiates between a single-byte character (half- width) and representation of the double-byte character of the same character.

## **Use of UPDATE\_STATISTICS command?**

UPDATE\_STATISTICS command is basically used when a large processing of data has occurred. If a large amount of deletions, any modification or Bulk Copy into the tables has occurred, it has to update the indexes to take these changes into account. UPDATE\_STATISTICS updates the indexes on these tables accordingly.

**Updating statistics** ensures that queries compile with up-to-date statistics. However, updating statistics causes queries to recompile. We recommend not updating statistics too often because there is a performance tradeoff between improving query plans and the time it takes to recompile queries. The specific tradeoffs depend on your application. UPDATE STATISTICS can use tempdb to sort the sample of rows for building statistics.

**When is the UPDATE\_STATISTICS command used?**

This command is used, ones the processing of large data is done.  
When we delete a large number of files, alteration or reproduction takes place in the tables, to be concerned of these changes we need to restructure the indexes This is done **UPDATE\_STATISTICS**.

## **STUFF, COALESCE, FOR XML PATH**

**STUFF(string, start, length, new-string):**

The STUFF() function deletes a part of a string and then inserts another part into the string, starting at a specified position. STUFF is used to replace the part of string with some other string OR It delete a specified length of characters within a string and replace with another set of characters.

**REPLACE(string, old-string, new-string):**

The REPLACE() function replaces all occurrences of a substring within a string, with a new substring. The search is case-insensitive.

**SUNSTRING(string, start, length):**

The SUBSTRING() function extracts some characters from a string.

**COALESCE(Val1, Val2, Val3, …):**

The COALESCE() function returns the first non-null value in a list.

**FOR XML PATH():**

The Path mode with FOR XML in [SQL Server](https://www.tutorialgateway.org/sql/) returns result set as the XML element.

Eg: SELECT ',' + name

FROM temp1

FOR XML PATH ('')

**IIF(condition, value-if-true, value-if-false):**

The IIF() function returns a value if a condition is TRUE, or another value if a condition is FALSE. Only Condition is required.

**NULLIF(expr1, expr2):**

The NULLIF() function returns NULL if two expressions are equal, otherwise it returns the first expression.

**ISNULL(expression, value):**

The ISNULL() function returns a specified value if the expression is NULL. If the expression is NOT NULL, this function returns the expression.

**LEN(string):**

Returns the number of characters of the specified string expression, excluding trailing blanks.

**DATALENGTH(string):**

Returns the number of bytes used to represent any expression including leading and trailing spaces. When processing a unicode string, DATALENGTH() will return twice the number of characters. This is because a unicode string stores 2 bytes per character. If result is stored as, **char(25)**, DATALENGTH() will return exactly 25 characters.

**GROUPING SETS:**

The GROUPING SETS defines multiple grouping sets in the same query.

SELECT column1, column2, aggregate\_function (column3)

FROM  table\_name

GROUP BY   GROUPING SETS ( (column1, column2),  (column1),  (column2), () );

**CUBE:**

The CUBE is a subclause of the [GROUP BY](http://www.sqlservertutorial.net/sql-server-basics/sql-server-group-by/) clause that allows you to generate multiple grouping sets. The following illustrates the general syntax of the CUBE:

SELECT d1,   d2,   d3,   aggregate\_function (c4)

FROM    table\_name

GROUP BY    CUBE (d1, d2, d3);

The CUBE generates all possible grouping sets based on the dimension columns d1, d2, and d3 that you specify in the CUBE clause. If you have N dimension columns specified in the CUBE, you will have 2N grouping sets.

**How can a SQL Developer prevent T-SQL code from running on a production SQL Server?**

Use IF logic with the **@@SERVERNAME function** compared against a string with a RETURN command before any other logic.

**What does it mean to have QUOTED\_IDENTIFIER ON? What are the implications of having it OFF?**

When **SET QUOTED\_IDENTIFIER** is **ON,** identifiers can be delimited by double quotation marks, and literals must be delimited by single quotation marks. When **SET QUOTED\_IDENTIFIER** is **OFF**, identifiers cannot be quoted and must follow all Transact-SQL rules for identifiers.

**Explain how to send email from SQL database.**

SQL Server has a feature for sending mails. Stored procedures can also be used for sending mail on demand. With SQL Server 2005, MAPI client is not needed for sending mails.  
**The following is the process for sending emails from database.**

* Make sure that the SQL Server Mail account is configured correctly and enable Database Mail.
* Write a script to send an e-mail. The following is the script.
* USE [YourDB]
* EXEC msdb.dbo.sp\_send\_dbmail
* @recipients = 'xyz@intellipaat.com; abc@intellipaat.com;pqr@intellipaat.com’
* @body = ' A warm wish for your future endeavor',
* @subject = 'This mail was sent using Database Mail' ;
* GO

**What is meant by incremental load?**

Incremental load refers to applying the dynamic changes as and when required in a specific period and predefined schedules.

**What is initial load and what is full load?**

In ETL the initial load is the process for populating all data warehousing tables for very first time. Full load means when the data is loaded for the first time all set records are loaded at a stretch depending on its volume. It would erase all contents in the table and would reload fresh data.

## **Partitioning in ETL?**

Partitioning in ETL refers to sub division of the transactions in order to improve their performances.

To improve the performances of ETL transactions the session partition is used. Power Mart does not allow sessions partition.

Increase in the number of partitions enables the informatics Server to create multiple connections to a host of sources.

Types of partitions in ETL are Round-Robin partition and Hash partition.

**What is Round Robin partitioning?**

In Round Robin partitioning the data is evenly distributed by the informatica among all the partitions. It is used when the number of rows in process in each of the partitions is nearly the same.

**What is Hash partitioning?**

In Hash partitioning the informatica server would apply a hash function in order to partition keys to group data among the partitions. It is used to ensure the processing of group of rows with the same partitioning key in same partition.

**What is meant by Worklet in ETL?**

Worklet is the set of tasks in ETL. It can be any set of tasks in the program.

**What is workflow in ETL?**

Workflow is a set of instruction that specifies the way of executing the tasks to the informatica.

**What is referred by Mapplet in ETL?**

Mapplet in ETL is used for the purpose of creation as well as configuration of a group of transformations.

## **Analytical function**

Analytical functions are one of the most popular tools among BI/Data analysts for performing complex data analysis. These functions perform computations over multiple rows and return the multiple rows as well. Aggregate and Analytic functions in SQL Server operate on a set of rows. However, unlike such aggregate functions as sum, count and average that return scalar values, analytic functions return a group of rows that can be further analyzed.

Few of the frequently used are:

* **CUME\_DIST**: the cume\_dist function is used to find the cumulative distribution of a value among the group of values
* **FIRST\_VALUE**: retrieves the first value from the specified column for the records that have been sorted using the ORDER BY clause
* **LAST\_VALUE**: reverse of the first value function and returns the last value for specific column from the ordered sequence
* **LEAD**: retrieve the values from next N rows. The column from which the value is to be fetched and the number of rows to be offset is specified inside the lead function.
* **LAG**: reverse of the lead function and is used to retrieve the values from previous N rows. The column from which the value is to be fetched and the number of rows to be offset is specified inside the lag function

Syntax:

analytic\_function\_name([argument\_list])  
OVER (  
[PARTITION BY partition\_expression,…]  
[ORDER BY sort\_expression, … [ASC|DESC]])

## **Windows function**

A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result. Window functions operate on a set of rows and return a single aggregated value for each row. The term Window describes the set of rows in the database on which the function will operate. We define the Window (set of rows on which functions operates) using an OVER() clause.

Types of Windows function:

* Aggregate Window Functions  
  SUM(), MAX(), MIN(), AVG(). COUNT()
* Ranking Window Functions  
  RANK(), DENSE\_RANK(), ROW\_NUMBER(), NTILE()
* Value Window Functions  
  LAG(), LEAD(), FIRST\_VALUE(), LAST\_VALUE()

Eg: SELECT duration\_seconds,

SUM(duration\_seconds) OVER (ORDER BY start\_time) AS running\_total

FROM tutorial.dc\_bikeshare\_q1\_2012

## **What is "scheduled jobs" or "scheduled tasks"?**

Scheduled tasks let you manage the tasks in an automated manner that runs on regular or predictable cycles. You can schedule administrative tasks and also determine the order in which tasks will run.

## **What is the DBCC command and why is it used**

DBCC stands for database consistency checker. This command is used to check the consistency of the database. DBCC command help to review and monitoring the maintenance of tables, database, and for validation of operations done on the database, etc. For example:

**DBCC CHECKDB**: It makes sure that table in the database and the indexes are correctly linked.

**DBCC CHECKALLOC**: It checks all pages in the database and makes sure that all are correctly allocated.

**DBCC CHECKFILEGROUP**: It checks all table file group for any damage.

If the user executes the above commands, a database snapshot is created through the database engine, and it continues in the consistent transactional state. After that, it runs the checks against stored database snapshot, and after the completion of the command, it dropped the snapshot.

## **What are the most common trace flags used with SQL Server?**

Trace flag in SQL server sets the specific characteristic of the server. It works as an "IF" condition for the SQL Server. The most common trace flags used with SQL Server are:

* Deadlock Information: 1204, 1205, 1222
* Network Database files: 1807
* Log Record for Connections: 4013
* Skip Startup Stored Procedures: 4022
* Disable Locking Hints: 8755
* Do Force uniform extent allocations instead of mixed page allocations 1118 (SQL 2005 and 2008).

## **Encryption mechanisms in SQL server?**

We can use encryption for security of data in the database in SQL Server. Following are the encryption mechanism used in SQL server:

* Transact-SQL functions
* Asymmetric keys
* Symmetric keys
* Certificates
* Transparent Data Encryption

## **Relational Algebra?**

Relational Algebra is a Procedural Query Language which contains a set of operations that take one or two relations as input and produce a new relationship. Relational algebra is the basic set of operations for the relational model. The decisive point of relational algebra is that it is similar to the algebra which operates on the number.

There are few fundamental operations of relational algebra:

* select
* project
* set difference
* union
* rename,etc.

## **Relational Calculus?**

Relational Calculus is a Non-procedural Query Language which uses mathematical predicate calculus instead of algebra. Relational calculus doesn't work on mathematics fundamentals such as algebra, differential, integration, etc. That's why it is also known as predicate calculus.

There is two type of relational calculus:

* Tuple relational calculus
* Domain relational calculus

## **What are the integrity rules in DBMS?**

Data integrity is one significant aspect while maintaining the database. So, data integrity is enforced in the database system by imposing a series of rules. Those set of integrity is known as the integrity rules.

There are two integrity rules in DBMS:

**Entity Integrity**: It specifies that "Primary key cannot have a NULL value."

**Referential Integrity**: It specifies that "Foreign Key can be either a NULL value or should be the Primary Key value of other relation

## **Rank v/s Dense\_Rank**

Rank does not gives a contious ranking if there are multiple records for the same rank while dense\_Rank gives a continuous ranking to the records. DENSE\_RANK() and RANK() differ on the point that in the former we get consecutive ranks while in the later the rank after a tie is skipped. For example, ranking using DENSE\_RANK() would be something like (1,2,2,3) whereas ranking using RANK() would be (1,2,2,4).

## **Windows Function v/s Aggregate Function**

All database users know about regular aggregate functions which operate on an entire table and are used with a GROUP BY clause. But very few people use Window functions in SQL. These operate on a set of rows and return a single aggregated value for each row.

The main advantage of using Window functions over regular aggregate functions is: Window functions do not cause rows to become grouped into a single output row, the rows retain their separate identities and an aggregated value will be added to each row.

## **Sub-type v/s Super-type entity**

Let’s say we have these different entities – vehicle, car, bike, Economy car, family car, sports car. Here, a vehicle is a super type entity. Car and bike are its sub-type entities. Furthermore, economy car, sports car, and family car are sub-type entities of its super-type entity- car. A super-type entity is the one which is at a higher level. Sub-type entities are ones which are grouped together on the basis of certain characteristics. For example, all bikes are two-wheelers and all cars are four wheelers. And since both are vehicles, so their super-type entity is ‘vehicle’.

## **Dimension v/s Attribute**

In short, dimensions are nothing but which represents qualitative data. For example data like a plan, product, class are all considered as dimensions. A dimension table contains descriptive or textual attributes.

The attribute is nothing but a subset of a dimension. Within a dimension table, we will have attributes. The attributes can be textual or descriptive. For example, product name and product category are nothing but an attribute of product dimensions.

## **Temp tables v/s Table variable**

Although both SQL Server temp tables and table variables are stored in the TempDB database, there are many differences between them such as:

* SQL temp tables are created using CREATE TABLE T-SQL statement, but table variables are created using DECLARE @name Table T-SQL statement.
* You can ALTER the SQL Server temp tables after creating it, but table variables don’t support any DDL statement like ALTER statement.
* SQL temp tables can’t be used in User Defined Functions, but table variables can be.
* SQL Server temporary tables honor explicit transactions defined by the user, but table variables can’t participate in such transactions.
* SQL temp tables support adding clustered and non-clustered indexes after the SQL Server temp table creation and implicitly by defining Primary key constraint or Unique Key constraint during the tables creation, but table variables support only adding such indexes implicitly by defining Primary key constraint or Unique key constraint during tables creation.
* SQL temp tables can be dropped explicitly, but table variables can’t be dropped explicitly, taking into consideration that both types are dropped automatically when the session in which they are created is disconnected.
* SQL Server temp tables can be local temporal tables at the level of the batch or stored procedure in which the table declared or global temporal tables where it can be called outside the batch or stored procedure scope, but table variables can be called only within the batch or stored procedure in which it is declared.

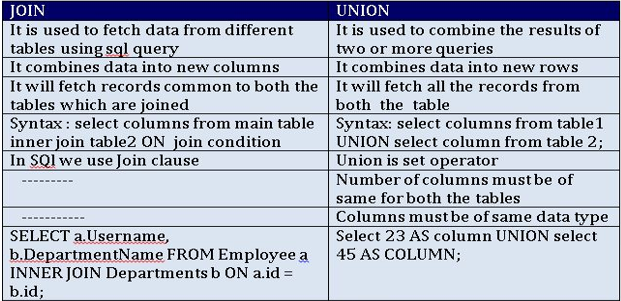
## **HAVING v/s WHERE**

1. HAVING is used to specify a condition for a group or an aggregate function used in select statement. The WHERE clause selects before grouping. The HAVING clause selects rows after grouping.
2. Condition specified in WHERE clause is used while fetching data (rows) from table, and data which doesn't pass the condition will not be fetched into result set, on the other hand HAVING clause is later used to filter summarized data or grouped data.
3. WHERE clause is used for filtering rows and it applies on each and every row, while HAVING clause is used to filter groups in SQL

## **JOIN v/s UNION?**

UNION puts lines from queries after each other, while JOIN makes a cartesian product and subsets it. The **JOIN** clause combines the attributes of two relations to form the resultant tuples whereas, **UNION** clause combines the result of two queries.

JOIN is applicable when the two involved relations have at least one common attribute and when the number of columns present in query are same and the corresponding attributes has the same domain.



## **JOIN v/s Nested Query? More efficient?**

* Subqueries can return single value or row set while join return rows.
* In most cases JOINs are faster than sub-queries as server is able to optimize it, there strength is higher readability. It is very rare for a sub-query to be faster.
* In JOINs RDBMS can create an execution plan that is better for your query and can predict what data should be loaded to be processed and save time, unlike the sub-query where it will run all the queries and load all their data to do the processing.
* The good thing in sub-queries is that they are more readable than JOINs: that's why most new SQL people prefer them; it is the easy way; but when it comes to performance, JOINS are better in most cases even though they are not hard to read too.

## **Semi Join v/s Anti Join in Oracle**

While a semi-join returns one copy of each row in the first table for which at least one match is found, an anti-join returns one copy of each row in the first table for which no match is found.

## **Identity v/s Sequence?**

In SQL Server, both the SEQUENCE object and IDENTITY property are used to generate a sequence of numeric values in ascending order.

The IDENTITY property is tied to a particular table and cannot be shared among multiple tables since it is a table column property.

On the flip side the SEQUENCE object is defined by the user and can be shared by multiple tables since is it is not tied to any table.

To create a Sequence object:

CREATE SEQUENCE [dbo].[SequenceCounter]

AS INT

START WITH 1

INCREMENT BY 1

To add a record using the same Sequence object:

INSERT INTO Cars3 VALUES (NEXT VALUE FOR [dbo].[SequenceCounter], 'C500','Merc',5000)

## **Identity v/s GUID?**

Identity (or AutoNumber) is a column that automatically generates numeric values. A start and increment value can be set, but most DBA leave these at 1. A GUID column also generates numbers; the value of this cannot be controlled. Identity/GUID columns do not need to be indexed.

GUID is a 16 byte binary SQL Server data type that is globally unique across tables, databases, and servers. GUIDs can be considered as global primary keys. Local primary keys are used to uniquely identify records within a table. On the other hand, GUIDs can be used to uniquely identify records across tables, databases, and servers. The term GUID stands for Globally Unique Identifier and it is used interchangeably with UNIQUEIDENTIFIER.

To create a GUID in SQL Server, the NEWID() function is used as shown below - ***SELECT NEWID()*.** Execute the above line of SQL multiple times and you will see a different value every time. This is because the NEWID() function generates a unique value whenever you execute it. To declare a variable of type GUID, the keyword used is UNIQUEIDENTIFIER as mentioned in the script below - ***DECLARE @UNI UNIQUEIDENTIFIER***

To use the GUID as primary key, we change the data type of the Id column from INT to *UNIQUEIDENTIFIER*. To set a default value for the column we will use the *default* keyword and set the default value as the value returned by the *‘NEWID()*’ function. This will ensure that whenever a new record is inserted in the EngStudents1 table, by default, the NEWID() function generates a unique value for the Id column. When inserting the records, we simply have to specify “*default*” as value for the first column. This will insert a default unique value to the Id column.

CREATE TABLE EnglishStudents1

(

Id UNIQUEIDENTIFIER PRIMARY KEY default NEWID(),

StudentName VARCHAR (50)

)

GO

INSERT INTO EnglishStudents1 VALUES (default,'Shane')

INSERT INTO EnglishStudents1 VALUES (default,'Jonny')

## **Materialized v/s Dynamic view?**

Materialized views

* Disk based and are updated periodically based upon the query definition.
* A materialized table is created or updated infrequently and it must be synchronized with its associated base tables.

Dynamic views

* Virtual only and run the query definition each time they are accessed.
* A dynamic view may be created every time that a specific view is requested by the user.

1. In Views query result is not stored in the disk or database but Materialized view allow to store the query result in disk or table.
2. When we create a view using any table, rowid of view is same as the original table but in case of Materialized view rowid is different.
3. In case of View we always get latest data but in case of Materialized view we need to refresh the view for getting latest data.
4. Performance of View is less than Materialized view. Materialized View responds **faster** than View as the Materialized View is precomputed.
5. In case of view, its only the logical view of table no separate copy of table but in case of Materialized view we get physically separate copy of table.
6. In case of Materialized view, we need an extra trigger or some automatic method so that we can keep MV refreshed, this is not required for views in the database.
7. Views are **not stored** physically on the disk. On the other hands, Materialized Views are **stored** on the disk.
8. Views can be defined as a **virtual table** created as a result of the query expression. However, Materialized View is a **physical copy**, picture or snapshot of the base table.
9. A view is always **updated** as the query creating View executes each time the View is used. On the other hands, Materialized View is updated **manually** or by applying **triggers** to it.
10. Materialized View **utilizes** the **memory space** as it stored on the disk whereas, the View is just a **display** hence it do not require memory space.
11. Materialized View responds faster as compared to View. But View always provides up to date information to the user.

## **Surrogate v/s Natural keys**

| **Surrogate Key** | **Natural Key** |
| --- | --- |
| The surrogate key has no business intelligence built into it. Meaning you cannot derive any meaning, or relationship between the surrogate key and the rest of the data columns in a row |  |
| Surrogate keys will not be updated over time. By using a surrogate key instead of a natural key in all foreign key relationships, we don’t have to update it. | Requires much more work to change a natural key, especially when foreign relationship have been built off the natural key.  If your business rules change, which would require you to update your natural key this can be done easily without causing a cascade effect across all foreign key relationships. |
| Surrogate keys are typically integers, which only require 4 bytes to store, so the primary key index structure will be smaller in size than their natural key counter parts.  Having a small index structure means better performance for JOIN operations. | Your primary key index will be larger because natural keys are typically larger in size then surrogate keys and are strings instead of integers joins between two tables on a natural key will take more time. |
| Surrogate keys are typically not useful when searching for data since they have no meaning. | Easier to search because natural keys have meaning and will be stored in your table. Without the natural key in your table, a search for records based on a natural key would require a join to the foreign key table to get the natural key. |
| If foreign key tables use surrogate keys, then you will be required to have a join to retrieve the real foreign key value. | Whereas if the foreign key table used a natural key then the natural key would be already be included in your table and no join would be required. Of course, this is only true if you only needed the natural key column returned in your query. |

## **Super v/s Candidate Key?**

| **S.NO** | **SUPER KEY** | **CANDIDATE KEY** |
| --- | --- | --- |
| 1. | Super Key is an attribute (or set of attributes) that is used to uniquely identifies all attributes in a relation. | Candidate Key is a proper subset of a super key. |
| 2. | All super keys can’t be candidate keys. | But all candidate keys are super keys. |
| 3. | Various super keys together make the criteria to select the candidate keys. | Various candidate keys together make the criteria to select the primary keys. |
| 4. | In a relation, number of super keys are more than number of candidate keys. | While in a relation, number of candidate keys are less than number of super keys. |
| 5. | Super key’s attributes can contain NULL values. | Candidate key’s attributes can also contain NULL values. |

## **Shared lock v/s Exclusive lock**

**Shared lock**: Shared lock is required for reading a data item. In the shared lock, many transactions may hold a lock on the same data item. When more than one transaction is allowed to read the data items then that is known as the shared lock.

**Exclusive lock**: When any transaction is about to perform the write operation, then the lock on the data item is an exclusive lock. Because, if we allow more than one transaction then that will lead to the inconsistency in the database.

## **Locking v/s Blocking v/s Deadlocking**

* **Locking:**Locking occurs when a connection needs access to a piece of data in database and it locks it for certain use so that no other transaction is able to access it.
* **Blocking:**Blocking occurs when a transaction tries to acquire an incompatible lock on a resource that another transaction has already locked. The blocked transaction remains blocked until the blocking transaction releases the lock.
* **Deadlocking:**Deadlocking occurs when two or more transactions have a resource locked, and each transaction requests a lock on the resource that another transaction has already locked. Neither of the transactions here can more forward, as each one is waiting for the other to release the lock.

## **Extension v/s Intension?**

**Extension:** The Extension is the number of tuples present in a table at any instance. It changes as the tuples are created, updated and destroyed. The actual data in the database change quite frequently. So, the data in the database at a particular moment in time is known as extension or database state or snapshot. It is time dependent.

**Intension:** Intension is also known as Data Schema and defined as the description of the database, which is specified during database design and is expected to remain unchanged. The Intension is a constant value that gives the name, structure of tables and the constraints laid on it.

## **Row oriented v/s Columnar database**

In Row oriented databases, indexes can be created but data is rarely stored in multiple sort orders. However, in Column oriented databases you can have the data stored in an arbitrary number of ways.

Traditional databases are row oriented databases that store data by row. The fields for each record are sequentially stored in a long row. For example, “Customer 1: name, address, date of birth, etc.” Then all the information for Customer 2 appears in a new row. In a columnar database, the names of every customer appear in a “name” column and all the addresses appear in an “address” column, etc.

A relational database is ideal for transactional applications because it stores rows of data. A columnar database is preferred for analytical applications because it allows for fast retrieval of columns of data. Columnar databases are designed for data warehousing and [big data processing](https://www.omnisci.com/platform/render) because they scale using distributed clusters of low-cost hardware to increase throughput.

The columnar database architecture has been called the [future of business intelligence (BI)](https://www.omnisci.com/learn/vast-world-of-iot) because it allows for instant analytical queries that enterprises depend on for business decisions. A columnar database is faster and more efficient than a traditional database because the data storage is by columns rather than by rows. A columnar database provides access to the most relevant elements, which increases the speed of a query even in a database containing millions of records.

Columnar databases are used in data warehouses where businesses send massive amounts of data from multiple sources for BI analysis. Column oriented databases have faster query performance because the column design keeps data closer together, which reduces seek time.

Columnar data is not ideal when you need to view multiple fields from each row. Traditional row databases tend to be better for queries seeking user-specific values only. Columnar databases can also take more time to write new data because column has to be written one by one.

Columnar databases excel at:

* Queries that involve only a few columns
* Aggregation queries against vast amounts of data
* Column-wise compression

Traditional databases are better for:

* Incremental data loading
* Online Transaction Processing (OLTP) usage
* Queries against only a few rows

## **Row store v/s Column store Indexes?**

Rowstore indexes perform best on queries that seek into the data, when searching for a particular value, or for queries on a small range of values. Use rowstore indexes with transactional workloads because they tend to require mostly table seeks instead of table scans.

Columnstore indexes give high performance gains for analytic queries that scan large amounts of data, especially on large tables. Use columnstore indexes on data warehousing and analytics workloads, especially on fact tables, because they tend to require full table scans rather than table seeks.

* Rowstore — The rowstore index is the traditional style that has been around since the initial release of SQL Server. Rowstore indexes are designed to speed the retrieval of data by enabling queries to quickly locate data by index rather than scanning an entire table. Rowstore data is logically organized by rows and columns, and is physically stored in row-oriented data pages. SQL Server internally organizes rowstore indexes using a B-Tree structure over the data pages. SQL Server supports clustered and non-clustered indexes. With clustered indexes, the data in the base table is organized according to the clustered index. Rowstore indexes perform best on queries that seek data by searching for a particular value or retrieving a small range of values. Rowstore indexes are a good fit for transactional workloads since these workloads tend to require table seeks instead of large-range table scans, and they often require frequent data updates.
* Columnstore — The columnstore index is also logically organized as a table with rows and columns, but the data is physically stored in a column-wise data format. Columnstore indexes work well for mostly read-only queries with large data sets, like data warehousing workloads. Columnstore indexes are not well-suited for queries that seek specific individual values. Columns often contain similar data which enables the data to be highly compressed, improving memory utilization and significantly reducing disk usage. A columnstore index can be clustered or non-clustered. Clustered and non-clustered columnstore indexes function the same. The difference is that a clustered columnstore index provides the primary storage for the entire table, while a nonclustered index is a secondary index that contains a copy of some of the columns in the underlying table. A clustered columnstore index can have one or more non-clustered B-tree indexes.

Overall, rowstore indexes tend to be better for online transaction processing (OLTP) workloads, which use more update and seek operations, while columnstore indexes tend to be better for online analytical processing (OLAP) workloads, which use more read operations. Rowstore indexes tend to be better at performing random reads and writes. Columnstore indexes tend to be better for performing sequential reads and writes.

## **Clustered v/s Non-Clustered Indexes?**

The differences between these two indexes is very important from SQL performance perspective.

1. One table can have only one clustered index, but it can have many non-clustered index. (Approximately 250).
2. A clustered index determines how data is stored physically in the table. Clustered index stores data in the cluster, related data is stored together, so that retrieval of data becomes simple.
3. Clustered indexes store the data information and the data itself whereas non-clustered index stores only the information, and then it will refer you to the data stored in clustered data.
4. Reading from a clustered index is much faster than reading from non-clustered index from the same table since they don’t involve any extra lookup step.
5. Clustered indexes only sort tables. Therefore, they do not consume extra storage. Non-clustered indexes are stored in a separate place from the actual table claiming more storage space.
6. Clustered index sort and store data row in the table or view based on their key value, while non-cluster has a structure separate from the data row.

A table without a clustered index is called a heap, due to its unordered structure. Data in a heap table isn’t sorted, usually the records are added one after another, as they are inserted into the table. They can also be rearranged by the database engine, but again, without a specific order. When you insert a lot of rows into a heap table, the new records are written on data pages without a specific order. Finding a record in a heap table can be compared to finding a specific leaf in a heap of leaves. It is inefficient and requires time. A heap can have one or several non-clustered indexes, or no indexes at all.

A clustered index organizes table data, so data is queried quicker and more efficiently. A clustered index consists of both index and data pages, while a heap table has no index pages; it consists only of data pages. In other words, it is not just an index, i.e. a pointer to the data row that contains the key value, but it also contains table data. The data in the clustered table is sorted using the values of the columns the clustered index is made of. Finding a record from a table with a proper clustered index is quick and easy like finding a name in an alphabetically ordered list. A general recommendation for all SQL tables is to have a proper clustered index

A non-clustered index is made only of index pages that contain row locators (pointers) for records in data pages. It doesn’t contain data pages, like clustered indexes. Clustered index is the index according to which data is physically stored on disk. Therefore, only one clustered index can be created on a given database table.  
Non-clustered indexes don’t define physical ordering of data, but logical ordering. Typically, a tree is created whose leaf point to disk records. [B-Tree](http://en.wikipedia.org/wiki/B-tree) or [B+ tree](http://en.wikipedia.org/wiki/B+_tree) are used for this purpose.

Every non-clustered index contains the key values you specify, plus the value obtained from the clustered index; this acts as a pointer to the actual rows when you need their data. This means that the query engine will never need to visit the actual rows—provided that your unclustered index is keyed on the values you need for a given task. It will learn everything it needs from the index and scope, and then it will visit the qualifying rows.

## **Normalization v/s Denormalization**

| **S.NO** | **NORMALIZATION** | **DENORMALIZATION** |
| --- | --- | --- |
| 1. | In normalization, Non-redundancy and consistency data are stored in set schema. | In denormalization, data are combined to execute the query quickly. |
| 2. | In normalization, Data redundancy and inconsistency is reduced. | In denormalization, redundancy is added for quick execution of queries. |
| 3. | Data integrity is maintained in normalization. | Data integrity is not maintained in denormalization. |
| 4. | Number of tables in normalization is increased. | Denormalization, Number of tables in decreased. |
| 5. | Normalization optimize the uses of disk spaces. | Denormalization do not optimize the disk spaces. |

## **Strong v/s Weak SQL?**

| **S.NO** | **STRONG ENTITY** | **WEAK ENTITY** |
| --- | --- | --- |
| 1. | Strong entity always has primary key. | While weak entity has partial discriminator key. |
| 2. | Strong entity is not dependent of any other entity. | Weak entity depends on strong entity. |
| 3. | Strong entity is represented by single rectangle. | Weak entity is represented by double rectangle. |
| 4. | Two strong entity’s relationship is represented by single diamond. | While the relation between one strong and one weak entity is represented by double diamond. |
| 5. | Strong entity have either total participation or not. | While weak entity always has total participation. |

## **Embedded v/s Dynamic SQL?**

Static or Embedded SQL

* SQL statements in an application that do not change at runtime and, therefore, can be hard-coded into the application.

Dynamic SQL

* SQL statements that are constructed at runtime; for example, the application may allow users to enter their own queries.
* Dynamic SQL is a programming technique that enables you to build SQL statements dynamically at runtime. You can create more general purpose, flexible applications by using dynamic SQL because the full text of a SQL statement may be unknown at compilation.
* Dynamic SQL can be used in these ways :
  1. Writing a query with parameters.
  2. Using EXEC.
  3. Using sp\_executesql.

| **S.No.** | **Static (embedded) SQL** | **Dynamic (interactive) SQL** |
| --- | --- | --- |
| 1. | In static SQL how database will be accessed is predetermined in the embedded SQL statement. | In dynamic SQL, how database will be accessed is determined at run time. |
| 2. | It is more swift and efficient. | It is less swift and efficient. |
| 3. | SQL statements are compiled at compile time. | SQL statements are compiled at run time. |
| 4. | Parsing, validation, optimization, and generation of application plan are done at compile time. | Parsing, validation, optimization, and generation of application plan are done at run time. |
| 5. | It is generally used for situations where data is distributed uniformly. | It is generally used for situations where data is distributed non-uniformly. |
| 6. | EXECUTE IMMEDIATE, EXECUTE and PREPARE statements are not used. | EXECUTE IMMEDIATE, EXECUTE and PREPARE statements are used. |
| 7. | It is less flexible. | It is more flexible. |

## **Fact table v/s Dimension table?**

The Fact Table and Dimension Table, are the essential factors to create a **schema**. A fact table’s  record is a combination of attributes from different dimension tables. Fact table helps the user to analyze the business dimensions which helps him in decision taking to improve his business. On the other hands, dimension tables help fact table to collect dimensions along which the measures has to be taken. Fact table contains measurement along the dimension/attributes of a dimension table.

The point that distinguishes Fact table and Dimension table is that the**dimension table** contains attributes along which measures are taken in **fact table**.

| **BASIS FOR COMPARISON** | **FACT TABLE** | **DIMENSION TABLE** |
| --- | --- | --- |
| Basic | Fact table contains the measurement along the attributes of a dimension table. | Dimension table contains the attributes along which fact table calculates the metric. |
| Attribute & Records | Fact table contains less attributes and more records. | Dimension table contains more attributes and less records. |
| Table size | Fact table grows Vertically. | Dimension table grows horizontally. |
| Key | Fact table contains a primary key which is a concatenation of primary keys of all dimension table. | Each dimension table contains its primary key. |
| Creation | Fact table can be created only when dimension tables are completed. | Dimension tables need to be created first. |
| Schema | A schema contains less number of fact tables. | A schema contains more number of dimension tables. |
| Attributes | Fact table can have data in numeric as well as textual format. | Dimension table always contains attributes in textual format. |

## **Star v/s Snowflake schema?**

The crucial difference between Star schema and snowflake schema is that star schema does not use normalization whereas snowflake schema uses normalization to eliminate redundancy of data.

| **BASIS FOR COMPARISON** | **STAR SCHEMA** | **SNOWFLAKE SCHEMA** |
| --- | --- | --- |
| Structure of schema | Contains fact and dimension tables. | Contains sub-dimension tables including fact and dimension tables. |
| Use of normalization | Doesn't use normalization. | Uses normalization and denormalization. |
| Ease of use | Simple to understand and easily designed. | Hard to understand and design. |
| Data model | Top-down | Bottom-up |
| Query complexity | Low | High |
| Foreign key join used | Fewer | Large in number |
| Space usage | More | Less |
| Time consumed in query execution | Less | More comparatively due to excessive use of join. |

**Star schema** is the simple and common modelling paradigm where the data warehouse comprises of a **fact table** with a single table for each dimension. The schema imitates a star, with **dimension table** presented in an outspread pattern encircling the central fact table. The dimensions in fact table are connected to dimension table through primary key and foreign key.

**Snowflake schema** is the kind of the star schema which includes the hierarchical form of dimensional tables. In this schema, there is a fact table comprise of various dimension and sub-dimension table connected across through primary and foreign key to the fact table. It is named as the snowflake because its structure is similar to a snowflake.

It uses normalization which splits up the data into additional tables. The splitting results in the reduction of redundancy and prevention from memory wastage. A snowflake schema is more easily managed but complex to design and understand. It can also reduce the efficiency of browsing since more joins will be required to execute a query.

1. Star schema contains just **one** dimension table for one dimension entry while there may exist dimension and sub-dimension table for one entry.
2. Normalization is used in snowflake schema which eliminates the data redundancy. As against, normalization is not performed in star schema which results in data redundancy.
3. Star schema is simple, easy to understand and involves less intricate queries. On the contrary, snowflake schema is hard to understand and involves complex queries.
4. The time consumed for executing a query in a star schema is less. Conversely, snowflake schema consumes more time due to the excessive use of joins.

If the purpose of your project is to do more of dimension analysis, you should go for snowflake schema. For example, if you need to find out that ***“how many subscribers are tied to a particular plan are currently active?”***– go with snowflake model.

If the purpose of your project is to do more of metrics analysis, you should go with a star schema. **For example,** if you need to find out that ***“what is the claim amount paid to a particular subscriber?”***– go with a star schema.

*In my project, we used snowflake schema because we had to do analysis across several dimensions and generate summary reports for the business. Another reason for using snowflake schema was it is less memory consumption.*

## **Varchar v/s Char?**

* CHAR and VARCHAR are differ in storage and retrieval.
* CHAR column length is fixed while VARCHAR length is variable.
* The maximum no. of character CHAR data type can hold is 255 character while VARCHAR can hold up to 4000 character.
* CHAR is 50% faster than VARCHAR.
* CHAR uses static memory allocation while VARCHAR uses dynamic memory allocation.

## **Varchar v/s Varchar2?**

varchar is an ANSI standard which can store upto 2000 bytes and occupies space for the NULL values.

**Varchar2**is an oracle standard which can store upto 4000 bytes does not occupy space for the NULL values.

## **Truncate v/s Delete?**

1. TRUNCATE is a DDL command, whereas DELETE is a DML command.
2. We can’t execute a trigger in case of TRUNCATE whilst with DELETE, we can accomplish a trigger.
3. TRUNCATE is quicker than DELETE, for the reason that when we use DELETE, at that time it stores the whole statistics in the rollback gap on or after where we can get the data back after removal. In case of TRUNCATE, it will not store data in rollback gap and will unswervingly rub it out. TRUNCATE do not recover the deleted data.
4. We can use any condition in WHERE clause using DELETE but it is not possible with TRUNCATE.
5. If a table is referenced by any foreign key constraints, then TRUNCATE won’t work.
6. If you use TRANSACTIONS in your code, TRUNCATE can be rolled back. If there is no transaction is used and TRUNCATE operation is **committed**, it can not be retrieved from log file. TRUNCATE is DDL operation and it is not logged in log file.
7. If the table to be truncated has an [identity column](http://www.sqlservertutorial.net/sql-server-basics/sql-server-identity/), the counter for that column is reset to the seed value when data is deleted by the TRUNCATE TABLE statement but not the DELETE statement.

The concept that a Truncate statement cannot be rolled back is fairly commonly misunderstood. When you execute a Truncate statement, it does not get logged in the log file as it is a DDL statement. So, if you Truncate a table, you cannot Roll Back to a point in time before the truncate.

However, in a Transaction, Rollback is permitted and functions just as any other rollback would. The rollback referred to is not transactional rollback. In essence: Anywhere before a COMMIT is issued, you can rollback a TRUNCATE statement. After a COMMIT, you cannot rollback the data even by using the log file. Unrecoverable.

## **Local v/s Global temp table**

**Local temp table:**

1. In SQL Server, local temporary tables are visible only in the current session. So if you create a local temporary table in one session, you cannot access it in other sessions.
2. Valid only for current connection and are deleted when the connection closes.
3. If a local temporary table created in a stored procedure, it is dropped automatically when the stored procedure is **finished**. This means that this local temporary table can be referenced only by **nested** stored procedures. The local temporary table **cannot** be referenced by the stored procedure or application that called the stored procedure that created the local temporary table.

**Global temp table:**

1. In SQL Server, global temporary tables are visible to all sessions (connections). So if you create a global temporary table in one session, you can start using it in other sessions.
2. Available to all connections once created.
3. Global temporary table is automatically dropped when the session that **created** the table ends and the last active Transact-SQL statement (**not session**) referencing this table in other sessions ends.

You cannot access local and global temporary tables in functions (UDFs)

## **CTE v/s Temp Table**

* CTEs are unindexable (but can use existing indexes on referenced objects) and cannot have constraints whereas Temp Tables are real materialized tables that exist in tempdb
* Temp table is a real object in tempdb, but cte is only a kind of wrapper around complex query to simplify syntax of organize recursion in one step.
* A temp table is good for re-use or to perform multiple processing passes on a set of data whereas CTE can be used either to recurse or to simply improved readability.  
  And, like a view or inline table valued function can also be treated like a macro to be expanded in the main query
* When data in the CTE is small and there is strong readability improvement as with the case in recursive tables. However, its performance is certainly no better than table variables and when one is dealing with very large tables, temporary tables significantly outperform CTE. This is because you cannot define indices on a CTE and when you have large amount of data that requires joining with another table (CTE is simply like a macro). If you are joining multiple tables with millions of rows of records in each, CTE will perform significantly worse than temporary tables.
* The temp table is stored in tempdb on SQL Server, which is disk but has the benefit of being indexed and SQL optimizer works well on select queries in that case. Not sure on which db or disk area the CTE is stored (when it exceeds the memory size and is queued for IO paging) but its never optimzed with the large volume of data. I have used the compiler option (with recompile) sometimes to make it faster.
* Temp tables are always on disk - so as long as your CTE can be held in memory, it would most likely be faster (like a table variable, too). But then again, if the data load of your CTE (or temp table variable) gets too big, it'll be stored on disk, too, so there's no big benefit.
* In general, I prefer a CTE over a temp table since it's gone after I used it. I don't need to think about dropping it explicitly or anything.
* As far as when to use each, they have very different use cases. If you will have a very large result set, or need to refer to it more than once, put it in a #temp table. If it needs to be recursive, is disposable, or is just to simplify something logically, a CTE is preferred.
* Also, a CTE should never be used for performance. You will almost never speed things up by using a CTE, because, again, it's just a disposable view. You can do some neat things with them but speeding up a query isn't really one of them.
* A CTE may be called repeatedly within a query and is evaluated every time it is referenced - this process can be recursive. If it is just referred once then it behaves much like a sub-query, although CTEs can be parameterised.

## **CTE v/s View**

CTE extensively use TempDb and if your TempDb is not properly configured or your CTE is dealing with large number of rows then it is not a best bet. Choose View when dealing with large amount of data and CTE with small. You can choose CTE also when you need recursion

## **Sub-Query v/s CTE**

Both **CTEs** and **Sub Queries** have pretty much the same performance and function. CTE’s have an advantage over using a subquery in that you can use recursion in a CTE. The biggest advantage of using CTE is readability. CTEs can be referenced multiple times in the same statement whereas sub query cannot.

There are some differences between subqueries and CTEs, notably:

* A subquery is defined within an outer query. A CTE is defined before calling it from within the query.
* A CTE can reference itself, a subquery cannot.
* A CTE can reference other CTEs within the same WITH clause (Nest). A subquery cannot reference other subqueries.
* A CTE can be referenced multiple times from a calling query. A subquery cannot be referenced.

## **Sub-Query v/s Correlated Query?**

**Subquery** – The inner query is executed only once. The inner query will get executed first and the output of the inner query used by the outer query. The inner query is not dependent on outer query.

**Correlated subquery**: – The outer query will get executed first and for every row of outer query, inner query will get executed. So, the inner query will get executed as many times as number of rows in the result of the outer query. The outer query output can use the inner query output for comparison. This means inner query and outer query dependent on each other.

**Correlated subqueries** are the one in which **inner query or subquery reference outer query**. Outer query needs to be executed before inner query. One of the most common *example of correlated subquery* is using keywords exits and not exits. An important point to note is that **correlated subqueries are slower queries** and one should avoid it as much as possible.

It’s most important to be attentive of the arrange of operations in an interrelated subquery.  
**First,** a row is processed in the outer doubt.  
**Then,** for that exacting row, the subquery is executed – as a result for each row processed by the outer query, the subquery will also be processed. In correlated subquery, each time a line is worked for Emp1, the subquery will also make a decision on the exacting row’s value for Emp1.Salary and run. And the outer query will move on to the next row, and the subquery will execute for that row’s value of Emp1.Salary.  
It will persist in anticipation of the **“WHERE (1) = (… )”** state is pleased.

**Non-Correlated subquery**: – In non-correlated subquery, **inner query doesn't depend on outer query** and can run as standalone query. Inner query executes before outer query. *Subquery used along-with IN or NOT IN sql clause is good examples of Non-correlated subquery in SQL*.

## **Triggers v/s Stored Procedure?**

A [Trigger](http://en.wikipedia.org/wiki/Database_trigger) is a code that associated with insert, update or delete operations. The code is executed automatically whenever the associated query is executed on a table. Triggers can be useful to maintain integrity in database.

A [stored procedure](http://en.wikipedia.org/wiki/Stored_procedure) is like a function that contains a set of operations compiled together. It contains a set of operations that are commonly used in an application to do some common database tasks.

Unlike Stored Procedures, Triggers cannot be called directly. They can only be associated with queries.

## **EXEC v/s sp\_executesql?**

**sp\_executesql (also known as “Forced Statement Caching”)**

* Executes a Transact-SQL statement or batch that can be reused many times, or one that has been built dynamically. The Transact-SQL statement or batch can contain embedded parameters.
* Allows for statements to be parameterized. Therefore it’s more secure than EXEC in terms of SQL injection.
* Creates a plan on first execution (similar to stored procedures) and subsequent executions reuse this plan. sp\_executesql can leverage cached query plans. The TSQL string is built only one time, after that every time same query is called with sp\_executesql, SQL Server retrieves the query plan from cache and reuses it.
* Only allows parameters where SQL Server would normally allow parameters; however, this string can be built using forms of dynamic constructs.
* *sp\_executesql* gives you the advantage of having a place holder and pass the actual value at **runtime.**
* Has strongly typed variables/parameters – and this can reduce injection and offer some performance benefits.

**EXEC  (also known as “Dynamic String Execution” or DSE)**

* Allows \*any\* construct to be built.
* Treats the statement similarly to an adhoc statement. This means that the statement goes through the same process that adHoc statements do – they are parsed, probably parameterized and possibly deemed “safe” for subsequent executions to re-use.
* Does not have strongly typed parameters in the adhoc statement and therefore can cause problems when the statements are executed.
* With *Exec* You can't have a **place holder** in your T-Sql statement string.
* Temp tables created in EXEC can not use temp table caching mechanism.
* Does not force a plan to be cached.
* This can be a pro in that SQL Server can create a plan for each execution.
* This can be a con in that SQL Server needs to recompile/optimize for each execution.

| **sp\_executesql** | **EXEC Command** |
| --- | --- |
| Reuses the cached plan | Generates multiple plans when executed with different parameters |
| Less prone to SQL Injection | Prone to SQL injection |
| Supports parameterization | Does not support parameterization |
| Supports output variable | Output variable is not supported |

## **Hashing v/s B-Trees for storing indexes in databases?**

**Hash Indexes**

* Hash Indexes are **suitable for point lookups.** By point lookups, I mean “equality” queries for a given key. For example, the following query will benefit from using a hash index and is likely to get amortized O(1) lookup cost if there is a hash index created on “SSN” column.
* “SELECT \* FROM T WHERE SSN=k”;
* The above point also implies that hash indexes are more suitable for single value/row lookups or **primary key based lookups**
* Let’s talk about NON-UNIQUE indexes using hash index with the following example query: SELECT \* FROM T WHERE LAST\_NAME=’ROOT’.
* Let’s assume that table T has 3 such rows.
* If all the 3 rows are stored in different disk blocks, then a hash index will require 3 I/Os to fetch result set. On the other hand, a 3 level B-Tree Index will require 6 I/Os : root block + branch block + leaf block + 3 I/Os on respective blocks whose information was retrieved from leaf block.
* Also read about **Oracle Hash Clusters** which are used to create hash indexes. Attempt is made to co-locate the data for a given cluster key. In other words, all the three rows for LAST\_NAME=’ROOT’ can be found in the same block, and this would require only a single I/O.
* Hash Indexes are **not at all suitable for range scans in the databases** since they do not store the keys in any order. If we plan to use hash indexes for range scans, the lookup through index structure effectively degenerates into searching the entire database, and thus rules out any benefit of creating index on the target column.
* Moreover, even if we plan to do this, hash index by itself is not going to return results in sorted order. It will require additional steps to first sort the matching data on the given key before returning the result set to user.
* Hash Indexes are **not suitable for prefix matching of the key** since the lookup operation is based on a hash computation of the key, and thus needs the full exact key.
* We need to be very **careful with scalability of hash index structure**. As we can imagine, the underlying data structure used to build a hash index is a hash table (aka hash map). The performance of lookup operation with hash index can degrade:
* There are lots of collisions leading to long overflow chains or whatever the strategy is for resolving collisions.
* The lookup cost will increase in such cases and will require to resize the underlying hash table by adding more buckets and rehashing the existing items.

**B-Tree Indexes**

* B-Tree Indexes are **suitable for range queries/range scans** since the keys are ordered. For example, the following types of queries will benefit from a typical B-Tree Index structure on “SALARY” column.
* “SELECT \* FROM T WHERE SALARY>5000 AND SALARY<10000”
* “SELECT \* FROM T WHERE SALARY>5000”;
* The result set for such queries will contain rows in sorted order since the index stores the keys in order.
* The above point about suitability of B-Tree Indexes for range scans is valid for both UNIQUE and NON-UNIQUE Indexes.
* It is not the case that B-Tree Indexes are not suitable point lookups. We can definitely use B-Tree indexes for point lookups but **hash indexes are usually more efficient for such workloads**.
* The reason is extra I/O done in B-Tree indexes from the root block -> branch block -> leaf block to get to the data of interest whereas the cost of same operation with hash index is likely to be amortized O(1) provided the hash index is properly sized and handles collisions appropriately.
* B-Tree Indexes are **efficient for both full-key and prefix-key matching** queries.
* B-Tree indexes can be kept of optimal height to minimize the number of I/Os and the scalability problem mentioned above for hash indexes is not really a concern for B-Tree indexes.

## **OLTP v/s OLAP**

This difference between OLAP and OLTP also gives you the way to choosing the design of schema. If your system is OLTP, you should go with star schema design and if your system is OLAP, you should go with snowflake schema.

* OLTP is the transaction system that collects business data. Whereas OLAP is the reporting and analysis system on that data.
* OLTP systems are optimized for INSERT, UPDATE operations and therefore highly normalized. On the other hand, OLAP systems are deliberately denormalized for fast data retrieval through SELECT operations.

| **OLAP (ONLINE ANALYTICAL PROCESSING)** | **OLTP (ONLINE TRANSACTION PROCESSING)** |
| --- | --- |
| Consists of historical data from various Databases. | Consists only operational current data. |
| It is subject oriented. Used for Data Mining, Analytics, Decision making, etc. | It is application oriented. Used for business tasks. |
| The data is used in planning, problem solving and decision making. | The data is used to perform day to day fundamental operations. |
| It reveals a snapshot of present business tasks. | It provides a multi-dimensional view of different business tasks. |
| Large amount of data is stored typically in TB, PB | The size of the data is relatively small as the historical data is archived. For ex MB, GB |
| Relatively slow as the amount of data involved is large. Queries may take hours. | Very Fast as the queries operate on 5% of the data. |
| It only need backup from time to time as compared to OLTP. | Backup and recovery process is maintained religiously |
| This data is generally managed by CEO, MD, GM. | This data is managed by clerks, managers. |
| Only read and rarely write operation. | Both read and write operations. |

## **Datawarehouse v/s Database**

| **Database** | **Datawarehouse** |
| --- | --- |
| It supports operational processes. | It supports analysis and performance reporting. |
| Capture and maintain the data. | Explore the data. |
| Current data. | Multiple years of history. |
| Data is balanced within the scope of this one system. | Data must be integrated and balanced from multiple system. |
| Data is updated when transaction occurs. Deletes, inserts, replaces and updates large numbers of short online transactions quickly. | Data is updated on scheduled processes. Rapidly analyze massive volumes of data and provide different viewpoints for analysts. |
| Data verification occurs when entry is done. | Data verification occurs after the fact. |
| 100 MB to GB. | 100 GB to TB. |
| ER based. | Star/Snowflake. |
| Application oriented. | Subject oriented. |
| Primitive and highly detailed. | Summarized and consolidated. |
| Flat relational. | Multidimensional. |
| OnLine Transaction Processing (OLTP) | OnLine Analytical Processing (OLAP) |
| Deletes, inserts, replaces and updates large numbers of short online transactions quickly. | Rapidly analyze massive volumes of data and provide different viewpoints for analysts. |
| Highly normalized data structure with many different tables containing no redundant data. Thus, data is more accurate but slow to retrieve. | Denormalized data structure with few tables containing repeat data. Thus, data is potentially less accurate but fast to retrieve. |
| Current, real-time data for one part of the business | Historical data for all parts of the business |
| Analysis is slow and painful due to the large number of table joins needed and the small time frame of data available. | Analysis is fast and easy due to the small number of table joins needed and the extensive time frame of data available. |
| Thousands of concurrent users supported. However, only one user can modify each piece of data at a time. | Small number of concurrent users. |
| Records data in an ACID-compliant manner to ensure the highest levels of integrity. | Not always ACID-compliant though some companies do offer it. |
| 99.99% uptime | Downtime is built-in to accommodate periodic uploads of new data |
| Limited to a single data source from a particular business function | All data sources from all business functions |
| Simple transactional queries | Complex queries for in-depth analysis |
| Highly granular and precise | As granular and precise as you want it to be |

## **Data Warehouse v/s Data Mart**

* Data marts are smaller subsets of data from a data warehouse.
* Data marts are a repository of essential data for a specific subgroup or use case where access can be restricted to that subgroup or use case. Only a few users have access to the entire data warehouse.
* Data marts are less expensive and can analyze data faster because they are smaller subsets of the data warehouse that is slower and overloaded.
* A data warehouse is significantly larger, generally a terabyte or more in size, where a data mart is usually less than 100 GB.
* Data warehouses contain all the filtered data for an entire enterprise and across multiple categories and organizations where a data mart has a limited range focused on one line of business.
* Multiple sources store data in a data warehouse, whereas only a few sources contribute data to a data mart.

## **Data Warehouse v/s Data Lake**

* A data lake stores all the data for the organization. A data warehouse will only store essential data for creating structured data models and reporting.
* Data lakes store the data forever so that enterprises can pull the data from any point in time for analysis.
* Data lakes utilize different hardware that allows for cost-effective terabyte and petabyte storage.
* Data warehouse extract data using quantitative metrics from transactional systems. A data lake will extract data from all data types, including non-traditional data types like web server logs, social network activity, sensor data, etc.
* Data warehouses are for operational users that need to generate reports for analytics. A data lake is for deep analysis that goes beyond the stored data of a data warehouse.
* Because data lakes store raw data, the can be accessed and search before it has been cleansed or structured, user can get results faster.
* In a data warehouse, the schema for the data is preset; that is, there is a plan for the data upon its entry into the database. In a data lake, this is not necessarily the case. A data lake can house both structured and unstructured data and does not have a predetermined schema. A data warehouse handles primarily structured data and has a predetermined schema for the data it houses.
* A data warehouse is usually a [relational database](https://searchdatamanagement.techtarget.com/definition/relational-database) housed on an enterprise mainframe server or the cloud, whereas a data lake is usually housed in a Hadoop environment or similar big data repository.
* Data warehouses are useful when there is a massive amount of data from operational systems that need to be readily available for analysis. Data lakes are more useful when an organization needs a large repository of data but does not have a purpose for all of it and can afford to apply a schema to it upon access.

| **Parameters** | **Data Lakes** | **Data Warehouse** |
| --- | --- | --- |
| Data | Data lakes store everything. | Data Warehouse focuses only on Business Processes. |
| Users | Users are Data Scientists | Users are Business Professionals |
| Processing | Mainly unprocessed raw data | Highly processed data. |
| Accessibility | Highly accessible and quick to update | More complicated and costly to make changes |
| Type of Data | It can be Unstructured, semi-structured and structured. | It is mostly in tabular form & structure. |
| Task | Share data stewardship | Optimized for data retrieval |
| Agility | Highly agile, configure and reconfigure as needed. | Compare to Data lake it is less agile and has fixed configuration. |
| Users | Data Lake is mostly used by Data Scientist | Business professionals widely use data Warehouse |
| Storage | Data lakes design for low-cost storage. | Expensive storage that give fast response times are used |
| Security | Offers lesser control. | Allows better control of the data. |
| Replacement of EDW | Data lake can be source for EDW | Complementary to EDW (not replacement) |
| Schema | Schema on reading (no predefined schemas) | Schema on write (predefined schemas) |
| Data Processing | Helps for fast ingestion of new data. | Time-consuming to introduce new content. |
| Data Granularity | Data at a low level of detail or granularity. | Data at the summary or aggregated level of detail. |
| Tools | Can use open source/tools like Hadoop/ Map Reduce | Mostly commercial tools. |

## **Data Mart v/s Data Lake**

* Data lakes contain all the raw, unfiltered data from an enterprise where a data mart is a small subset of filtered, structured essential data for a department or function.
* Data marts are very specific, allowing for fast, effective analytics of relevant summarized information. Data lakes are better for broader, deep analysis of raw data.
* Data lakes are more flexible and the data is stored for an indefinite time where a data mart is restricted and exists for shorter time frames.
* Data lakes have a central archive where data marts can be store in different user areas.

## **SQL v/s NoSQL**

When it comes to choosing a database the biggest decisions is picking a **relational (SQL)**or **non-relational (NoSQL)** data structure. While both the databases are viable options still there are certain key differences between the two that users must keep in mind when making a decision.

**The Main Differences:**

1. **Type –**  
   SQL databases are primarily called as Relational Databases (RDBMS); whereas NoSQL database are primarily called as non-relational or distributed database.
2. **Language –**  
   SQL databases defines and manipulates data based structured query language (SQL). Seeing from a side this language is extremely powerful. SQL is one of the most versatile and widely-used options available which makes it a safe choice especially for great complex queries. But from other side it can be restrictive. SQL requires you to use predefined schemas to determine the structure of your data before you work with it. Also, all of your data must follow the same structure. This can require significant up-front preparation which means that a change in the structure would be both difficult and disruptive to your whole system.

A NoSQL database has dynamic schema for unstructured data. Data is stored in many ways which means it can be document-oriented, column-oriented, graph-based or organized as a KeyValue store. This flexibility means that documents can be created without having defined structure first. Also, each document can have its own unique structure. The syntax varies from database to database, and you can add fields as you go.

1. **The Scalability –**  
   In almost all situations SQL databases are vertically scalable. This means that you can increase the load on a single server by increasing things like RAM, CPU or SSD. But on the other hand NoSQL databases are horizontally scalable. This means that you handle more traffic by sharding, or adding more servers in your NoSQL database. It is similar to adding more floors to the same building versus adding more buildings to the neighborhood. Thus NoSQL can ultimately become larger and more powerful, making these databases the preferred choice for large or ever-changing data sets.
2. **The Structure –**  
   SQL databases are table-based on the other hand NoSQL databases are either key-value pairs, document-based, graph databases or wide-column stores. This makes relational SQL databases a better option for applications that require multi-row transactions such as an accounting system or for legacy systems that were built for a relational structure.
3. **Property followed –**  
   SQL databases follow ACID properties (Atomicity, Consistency, Isolation and Durability) whereas the NoSQL database follows the Brewers CAP theorem (Consistency, Availability and Partition tolerance).
4. **Support –**  
   Great support is available for all SQL database from their vendors. Also, a lot of independent consultations are there who can help you with SQL database for a very large scale deployments but for some NoSQL database you still have to rely on community support and only limited outside experts are available for setting up and deploying your large scale NoSQL deployments.  
     
   Some examples of SQL databases include PostgreSQL, MySQL, Oracle and Microsoft SQL Server. NoSQL database examples include Redis, RavenDB Cassandra, MongoDB, BigTable, HBase, Neo4j and CouchDB.

**Key highlights on SQL vs NoSQL:**

| **SQL** | **NOSQL** |
| --- | --- |
| RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS) | Non-relational or distributed database system. |
| These databases have fixed or static or predefined schema | They have dynamic schema |
| These databases are not suited for hierarchical data storage. | These databases are best suited for hierarchical data storage. |
| These databases are best suited for complex queries | These databases are not so good for complex queries |
| Vertically Scalable.  Vertical (scale-up with a larger server) | Horizontally scalable  Horizontal (scale-out across commodity servers) |

1. SQL databases are relational, NoSQL databases are non-relational.
2. SQL databases use structured query language and have a predefined schema. NoSQL databases have dynamic schemas for unstructured data.
3. SQL databases are vertically scalable, while NoSQL databases are horizontally scalable.
4. SQL databases are table-based, while NoSQL databases are document, key-value, graph, or wide-column stores.
5. SQL databases are better for multi-row transactions, while NoSQL is better for unstructured data like documents or JSON.

* SSMS is a Client tool to connect to database server.
* SQL Server runs on port 1433 and can be changed from the network utility TCP/IP properties.
* SQL Server has OLE-DB provider from Microsoft which allow linking to other services like Oracle.
* Checkpoints are those points to which the database engine can recover after a crash as a specified minimal point from where the transaction log record can be used to recover all the committed data up to the point of the crash.
* Relationship is defined as an association among two or more entities.
* Data abstraction in DBMS is a process of hiding irrelevant details from users.
* An entity set that doesn't have sufficient attributes to form a primary key is referred to as a weak entity set.
* Named pipes communicate across TCP port 445.
* Deadlock Information: 1204, 1205, 1222  
  Network Database files: 1807  
  Log Record for Connections: 4013  
  Skip Startup Stored Procedures: 4022  
  Disable Locking Hints: 8755  
  Forces uniform extent allocations instead of mixed page allocations 1118 – (SQL 2005 and 2008) To reduces TempDB contention.
* The sp\_MSforeachdb system stored procedure accepts the **@Command** parameter which can be exetecuted against all databases.
* OPENXML parses the XML data in SQL Server in an efficient manner.
* **CHARINDEX( findTextData, textData, [startingPosition] )** – Returns the starting position of the specified expression in a character string. The starting position is optional.
* **LEFT( character\_expression , integer\_expression )** – Returns the left part of a character string with the specified number of characters.
* **LEN( textData )** – Returns integer value of the length of the string, excluding trailing blanks.
* **LOWER ( character\_expression )** – Returns a character expression after converting uppercase character data to lowercase.
* **LTRIM( textData)** – Removes leading blanks.
* **PATINDEX( findTextData, textData )** – Returns integer value of the starting position of text found in the string.
* **REPLACE( textData, findTextData, replaceWithTextData )** – Replaces occurrences of text found in the string with a new value.
* **REPLICATE( character\_expression , integer\_expression )** – Repeats a character expression for a specified number of times.
* **REVERSE( character\_expression )** – Returns the reverse of a character expression.
* **RTRIM( textData)** – Removes trailing blanks. SPACE( numberOfSpaces ) – Repeats space value specified number of times.
* **STUFF( textData, start , length , insertTextData )** – Deletes a specified length of characters and inserts another set of characters at a specified starting point.
* **SUBSTRING( textData, startPosition, length )** – Returns portion of the string.
* **UPPER( character\_expression )** – Returns a character expression with lowercase character data converted to uppercase.
* **EXISTS**is used to check whether the result of a correlated nested query is empty (contains no tuples) or not.
* Fishbone diagram is called cause and effect diagram or Ishikawa diagram.
* The number of child tables that can be created that can be created out of the single parent table is equal to the number of fields/columns in the parent table that are non-keys.
* **Generally,** the fact table is in normalized form and dimension table is in de-normalized form.
* Metadata is data about data. It tells you that what kind of data is actually stored in the system, what is its purpose and for whom it is intended for.
* The sys.dm\_tran\_locks view returns a lot of information about currently active lock resources.
* **NOT FOR REPLICATION** indicates that when a record is replicated to this table, any value that is inserted into the identity column via the replication agent keep it's original value from the source system, but any records added locally still increment the identity value.
* A foreign key with **delete** **cascade** means that if a record in the parent table is deleted, then the corresponding records in the child table will automatically be deleted. This is called a cascade delete in SQL Server.

CREATE TABLE child\_table

(

column1 datatype [ NULL | NOT NULL ],

column2 datatype [ NULL | NOT NULL ],

...

CONSTRAINT fk\_name

FOREIGN KEY (child\_col1, child\_col2, ... child\_col\_n)

REFERENCES parent\_table (parent\_col1, parent\_col2, ... parent\_col\_n)

**ON DELETE CASCADE**

[ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]

);

* DELETE CASCADE option when creating foreign key constraints and how it helps keep the referential integrity of your database intact.
* A snapshot fact always show the latest (or last known) state of the measures. The latest records of a SCD type 2 dimension also do the same. The only difference is a fact shows the state of the measures whereas a SCD Type 2 table shows the state of the attributes.
* Windowed functions are defined in the ANSI spec to logically execute after the processing of GROUP BY, HAVING, WHERE. So it can only be used in SELECT and ORDER BY clause.
* Order of execution of SQL clauses – FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY, TOP.
* SSMS is a Client tool to connect to database server.
* SQL Server runs on port 1433 and can be changed from the network utility TCP/IP properties.
* SQL Server has OLE-DB provider from Microsoft which allow linking to other services like Oracle.
* Checkpoints are those points to which the database engine can recover after a crash as a specified minimal point from where the transaction log record can be used to recover all the committed data up to the point of the crash.
* Relationship is defined as an association among two or more entities.
* Data abstraction in DBMS is a process of hiding irrelevant details from users.
* An entity set that doesn't have sufficient attributes to form a primary key is referred to as a weak entity set.