Self-Study

1. Database Files

What are primary and secondary files in DB ?

Primary Data File:

The primary data file is the main file that contains the core data of the database, including system tables, user tables, indexes, and other database objects. It stores metadata about the database structure and serves as the starting point for the database so, There is exactly one primary data file per database. It has the **.mdf** file extension.

Secondary Data File:

Secondary data files are optional and used to store additional data to improve performance, scalability, or storage management. They can contain user-defined tables, indexes, or other objects, allowing data to be spread across multiple files for better management or performance. A database can have zero or more secondary data files. These files typically have the **.ndf** file extension.

What is the extension of .mdf , .ndf and .ldf files and what do they contain ?

.mdf (Primary Data File):

Stands for Master Data File. It stores the primary database data, including User data (tables, rows, etc.) , Database metadata (system catalog, schema information) , Indexes and other database objects. This is the core file required for the database to function.

.ndf (Secondary Data File):

Stands for .ndf (Named Data File). It stores additional user data or objects, such as tables and indexes, to distribute data across multiple files. Used to manage large databases by spreading data across multiple physical storage locations or to improve performance through parallel I/O operations.

.ldf (Log File):

Stands for Log Data File. It stores the transaction log, which records all database transactions (inserts, updates, deletes, etc.). It is used for Database recovery (restoring a database to a consistent state after a crash) , Transaction management , Replication and backup operations. A database typically has one or more log files, but at least one is required.

What are files and filegroups ?

A database is physically stored in files on disk, as described above (.mdf, .ndf, .ldf). Files are the physical storage units that hold the database's data and transaction logs. Each file is associated with a specific filegroup and resides on a specific disk or storage location.

Filegroups:

A filegroup is a logical structure that groups one or more database files together for organizational and administrative purposes.

Filegroups allow you to distribute data across multiple files or disks to improve performance, Manage backup and restore operations selectively (backing up only specific filegroups) and Place specific database objects (e.g., tables or indexes) in specific filegroups for better control over storage.

Types of Filegroups:

- **Primary Filegroup**: Contains the primary data file (.mdf) and possibly other secondary files (.ndf). It holds the system tables and metadata.
- **User-Defined Filegroups**: Created by the database administrator to store additional data files (.ndf) for specific tables or indexes.
- **Default Filegroup**: The filegroup where objects are stored if no specific filegroup is assigned. By default, this is the primary filegroup, but it can be changed.

Transaction log files (.ldf) are not part of any filegroup, as they are managed separately.

Benefits of Filegroups:

- **Performance**: Distribute data across multiple disks to reduce I/O bottlenecks.
- **Scalability**: Add more files to a filegroup as the database grows.
- **Maintenance**: Perform partial backups or restores on specific filegroups.
- **Organization**: Place frequently accessed tables or indexes on faster storage.

---

2. Normalization and Denormalization

What is normalization ?

Normalization is a database design process that organizes data to eliminate redundancy and ensure data integrity by structuring tables and relationships efficiently. It involves dividing a database into smaller, well-organized tables and defining relationships between them to minimize data anomalies during insertions, updates, and deletions.

Normalization is achieved by applying a series of rules, known as normal forms (NF), each building on the previous one. The process ensures that data is stored logically, dependencies are properly defined, and redundancy is reduced.

Normal Forms and Their Outputs:

First Normal Form (1NF):

> **Rule**: Eliminate repeating groups; ensure each column contains atomic (indivisible) values; each record is unique (primary key).
>
> **Output**: A table with no repeating groups, all attributes are atomic, and a primary key is defined.

Second Normal Form (2NF):

> **Rule**: Must be in 1NF; eliminate partial dependencies (non-key attributes must depend on the entire primary key, not just part of it).
>
> **Output**: A table where all non-key attributes are fully functionally dependent on the primary key, often resulting in separate tables for subsets of data.

Third Normal Form (3NF):

> **Rule**: Must be in 2NF; eliminate transitive dependencies (non-key attributes should not depend on other non-key attributes).
>
> **Output**: A table where non-key attributes depend only on the primary key, reducing redundancy by creating additional tables for transitive dependencies.

Boyce-Codd Normal Form (BCNF) (a stricter 3NF):

- **Rule**: Must be in 3NF; every determinant (attribute that determines another) must be a candidate key.
- **Output**: A table with no anomalies from functional dependencies, ensuring stricter integrity.

Higher Normal Forms (4NF, 5NF):

- **Rule**: Address multivalued dependencies (4NF) and join dependencies (5NF).
- **Output**: Highly specialized tables that eliminate complex dependencies, rarely needed in practice.

What is the difference between output of mapping and normalization ?

Mapping:

Mapping in databases refers to transforming data from one structure or format to another, such as converting data between ERD to an actual schema or during design processes.

The result of mapping is a new dataset or schema that aligns with the target structure, often preserving the original data's meaning but in a different format or model. For example, mapping might transform a flat file into a relational table or map data fields between two databases.

Normalization:

Normalization organizes data within a relational database to eliminate redundancy and ensure logical consistency. A set of tables aligning normal forms, with reduced redundancy, proper dependencies, and minimized data anomalies.

Differences:

Mapping is about transforming or aligning data between different structures (ERD and Schema); normalization is about optimizing a single database's internal structure.

Mapping produces a transformed dataset or schema; normalization produces a restructured set of tables with reduced redundancy and enforced dependencies.

They both will serve the same purpose at the end of the process but it is easier to follow mapping rules to produce more efficient design and the normalization forms are hard to achieve.

What is denormalization and In what business case do we need it ?

Denormalization is the process of intentionally introducing redundancy into a normalized database by combining tables or adding redundant data to improve read performance or simplify queries. It reverses some aspects of normalization to optimize for specific use cases, often at the cost of increased storage or potential data anomalies.

Business Cases for Denormalization

Denormalization is used in scenarios where read performance is critical, and the trade-offs of redundancy are acceptable. Common business cases include:

Data Warehousing and Reporting: In data warehouses, such as those used for business intelligence (BI) or analytics, queries often involve large-scale aggregations or joins across multiple tables. Denormalization (e.g., creating star or snowflake schemas) reduces the number of joins, speeding up complex queries for reports or dashboards. For example Combining customer and order details into a single table to avoid joins in frequent sales reports.

High-Traffic Web Applications: In applications with heavy read workloads (e.g., e-commerce platforms), denormalization reduces query complexity and latency, improving user experience. For example Storing a user's profile and recent activity in a single table to display their dashboard quickly.

Real-Time Systems: Systems requiring low-latency responses, such as recommendation engines or fraud detection, benefit from denormalized data to avoid complex joins. For example Storing product details and user preferences together for faster personalized recommendations.

Trade-Offs:

- **Increased Storage**: Redundant data consumes more disk space.
- **Update Complexity**: Redundant data requires careful management to maintain consistency during updates.
- **Write Performance**: Updates, inserts, or deletes may be slower due to multiple data copies.

---

3. Indexes

What are indexes ?

An index in a database is a data structure that improves the speed of data retrieval operations (SELECT queries) on a database table by providing a quick lookup mechanism. Indexes store a subset of a table's data (typically keys and pointers) in an optimized format, allowing the database engine to locate rows faster than scanning the entire table. They are stored in database files (.mdf or .ndf) and are critical for performance optimization.

What are the types of indexes ? What are the differences between these types ?

Clustered Index: A clustered index determines the physical order of data rows in a table on the disk. The table's data is sorted and stored based on the clustered index key. Only one clustered index per table is allowed because the table's data can only be physically sorted in one order. Usually created on the primary key by default (if not specified otherwise). Efficient for range queries, sorting, or queries using the indexed column (WHERE ID BETWEEN 100 AND 200).

Non-Clustered Index: A non-clustered index is a separate data structure that stores a copy of the indexed column(s) and pointers to the actual table rows (not the data itself). The table's physical order remains unchanged. Multiple non-clustered indexes can exist on a single table. Stored separately from the table data, often in the same filegroup but as a distinct structure. Smaller than clustered indexes because they store only the indexed columns and pointers. Improves performance for searches, joins, or filtering on non-primary key columns.

What is the meaning of clustered index in hard disk by primary key ?

A clustered index defines the physical order of data rows on the hard disk (or storage device, e.g., in .mdf or .ndf files in SQL Server). When a clustered index is created on a primary key:

The table's rows are physically sorted and stored on the disk based on the primary key values. For example, if the primary key is CustomerID, the rows are stored in ascending order of CustomerID. This means the data pages (stored in database files like .mdf) are organized to reflect the clustered index order, making lookups and range queries on the primary key very efficient.

What are the trade-offs for creating indexes ?

Creating indexes improves read performance but introduces trade-offs that must be carefully considered:

Pros:

- **Faster Reads:** Indexes significantly speed up SELECT queries, especially for filtering, sorting, or joining.
- **Efficient Data Retrieval:** Clustered indexes optimize range queries; non-clustered indexes improve specific column lookups.
- **Enforce Uniqueness:** Unique indexes ensure data integrity for columns like Email or SSN.

Cons:

- **Increased Storage:** Indexes consume additional disk space, especially non-clustered and covering indexes, which store copies of data (in .mdf or .ndf files).
- **Slower Writes:** Insert, update, and delete operations are slower because the database must update both the table and its indexes:
- **Maintenance Overhead:** Indexes require periodic maintenance (e.g., rebuilding or reorganizing) to prevent fragmentation, which impacts performance.
- **Increased Complexity:** Too many indexes can complicate query optimization, as the database engine must choose the best index, potentially leading to suboptimal plans.

Specific Trade-Offs by Index Type:

- **Clustered Index:** High write overhead due to physical reordering of data; limited to one per table.
- **Non-Clustered Index:** More storage and maintenance overhead; slower writes but allows multiple indexes.

When to Use Indexes:

- Create indexes on frequently queried columns (e.g., WHERE, JOIN, or ORDER BY clauses).

- Avoid over-indexing, as it can degrade write performance and increase storage needs.
- Monitor index usage and remove unused or rarely used indexes to reduce overhead.