- Changelog
- Postgres
- MySQL
- SQLite

Data structures in SQL: the definitive guide

Data structures designate different ways of storing data on a computer. Choosing the right data structure for your data can make a tremendous impact on your project.

5 months ago • 7 min read

By Felix Schildorfer

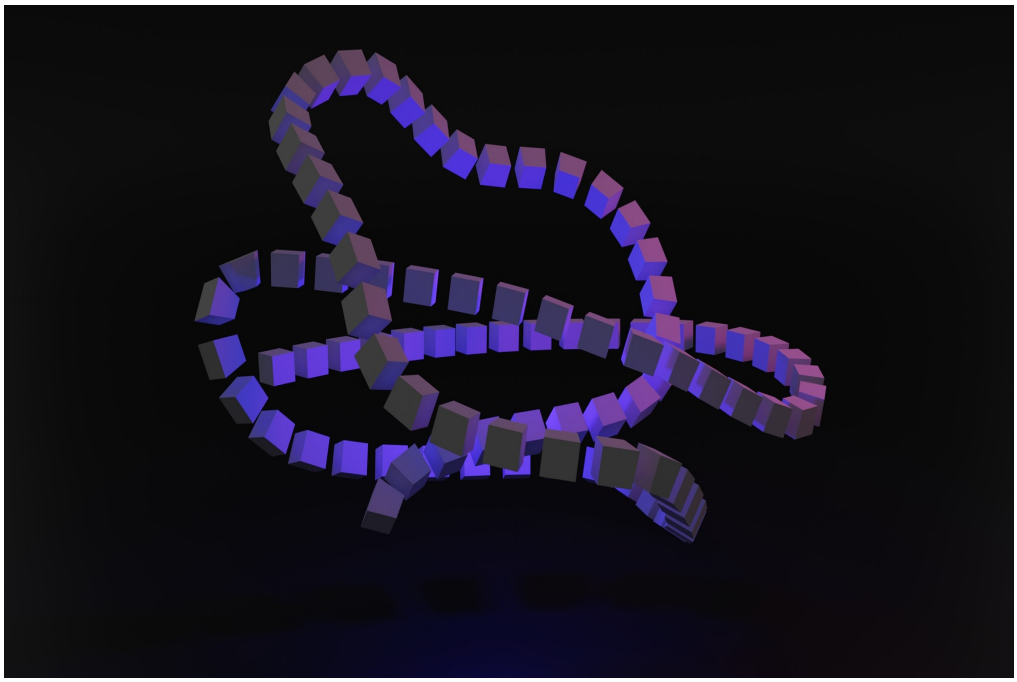


Photo by Shubham Dhage / Unsplash

Table of contents

Data structures designate different ways of storing data on a computer and form a vital part of any system or database design. The operations that you can perform on these data structures and the instructions you give to execute them, called algorithms, often have their essential functions tailored to the design of the data structure. Let's look at the importance of data structures and discuss their use in SQL databases.

Where are data structures used?

Apart from storing created data for data persistence, data structures also enable Core OS services and resources. For example, memory allocation, file directory management, and process scheduling can be done via linked lists, trees, and queues, respectively.

Developers can share packets via TCP/IP protocols organized through data structures. For example, efficient ordering and sorting methods are available for binary search trees, and priority queues allow programmers to manage items while respecting a set priority.

There are many easy ways to index and search your data in different data structures. Data structures also play a significant role in big data applications that ensure high performance and scalability.

How to choose a data structure

Different characteristics can help us categorize data structures. For example, they can be linear like an array, wherein data items appear in a predetermined order. Alternatively, they can be nonlinear like a graph, wherein the elements are unordered.

Homogeneous data structures need all elements to be of the same data type, while heterogeneous ones can house data of various types. In addition, data structures can either be static, with fixed sizes and memory locations, or dynamic, with sizes and memory locations that you can alter when necessary.

There is no straight answer to which data structure you should use. Depending on your use-case scenario, each data structure would have pros and cons. Therefore, it is crucial to consider the operations you would perform on the data while deciding which to use.

For instance, while it's easy to access any element of an array using its index, linked lists are better when you'd like to resize your elements. On the other hand, if you choose a data structure that is not suitable, your runtime could be prolonged, or your code could be unresponsive.

The five factors that developers commonly consider in picking data structures are:

- **Data Type:** the type of information you want to store
- **Use Case:** how you'll use the information
- **Location:** where the data is stored
- **Efficiency:** the best way for you to organize it for easy access
- **Storage:** how you can optimize storage reservation

Data structures in a SQL database management system

SQL, or Structured Query Language, is one of the most widely used programming languages for managing and organizing relational databases. An SQL database consists of tables with rows and columns. Developers use SQL to read, manipulate, access, and analyze their data to generate meaningful insights and make informed decisions.

These tables are database objects that can be considered containers. You can implement data structures like stacks, queues, and linked lists (introduced in the following sections) on a SQL server, a server with SQL databases.

Stack

You could implement a stack on an SQL server for several reasons. First, stacks take out memory, which is helpful in a front-end crash. You even use them to store an audit trail so that you can investigate a log when anything goes wrong.

Stacks can be of two types: LIFO and FIFO. You can operate a FIFO stack (First In First Out) from both ends, which means you can add or remove items from the front and back. On the other hand, you can only operate a LIFO stack (Last In First Out) from one end, which means you can store and retrieve items from the same side.

Removing an element from a stack is called popping, and adding an element to it is called pushing. These terms appear frequently, so they're helpful to learn.

Tree

Trees store items in a hierarchical manner. Each node in a tree is linked to a child node, a parent node, or both. The nodes are each associated with a key value, and they all descend from a single root node.

The SQL tree structure is helpful when the data you want to store in an SQL database has multiple levels. They save us the trouble of running several queries for each node to get to its child node. They also help us retrieve all the data by building the structure in code.

One common subtype of trees is binary trees. Search applications and expression solvers employ them frequently. Each node can have at most two child nodes in a binary tree.

Wide tables

Wide tables have a "sparse column" to store null values in an optimized manner. To make a table a wide table, all you have to do is add a column set to its definition. You can have up to thirty thousand columns in these tables.

System tables

You can store information about objects and instance configurations of the SQL server in system tables, which you can then access through system views. System tables are available in the master database and have the prefix "sys" in their names.

Partitioned tables

These tables have their data divided horizontally into units spread across one or more file groups in the database. This structure allows for easy management of large tables, as small chunks of the data can be accessed when necessary while maintaining the integrity of the complete data. By default, you can have up to fifteen thousand partitions in an SQL server.

Temporary tables (created in system database "tempdb")

If you need to share any data for a short period, you can use temporary tables. There are two kinds of temporary tables in the SQL server: local and global. The current user can only access local temporary tables within the existing connection to the database. If the connection is closed, these tables drop. However, you can access global temporary tables via any connection to the database since they are visible to any user when someone creates them. Global temporary tables only drop when the last connection using it closes.

Hash table

Hash tables are objects that contain key-value pairs. The key acts as an index for the values and makes the search of values very quick and convenient. Generation of the key happens through a hash function performing arithmetic functions. The "hash values" generated index a fixed-size hash table.

Only your current connection can access a hashtable that you create. A person connecting to the same database from a different connection cannot access it.

Heap table

A heap table is a form or organization of data with no clustered index, which means that the rows of data are stored randomly instead of in order. The data pages these rows are stored on are also in random order. When you need to store any new data, you'll add it to the first available, accessible location. If there's not enough space for the latest data on any of the pages, additional pages will automatically appear in the table, and you will insert the information there. Should you need to order the data in any way, you can use the SELECT statement's ORDER BY clause.

Graph

A graph in a SQL server is a collection of tables with nodes and edges. Graphs are trending because querying highly connected data can significantly improve performance. You can create one graph per database.

There are several flavors of graphs. A node table is a table composed of nodes of a similar type, and a collection of similar types of edges make up an edge table.

You can have both directed and undirected graph data structures. Directed graphs have edges that all point in a direction that show the start and end nodes. If there are no directions to indicate the start and end nodes, you have an undirected graph, and you can move in any direction. These graphs can also have nodes with no edges.

Developers use graphs to represent routes and locations in the mapping industry. The edges signify the routes, while the nodes represent the locations. In today's social media systems, every user can be considered a node. Edges form when the users connect.

For instance, a `Person` node table would have all `Person` nodes of a graph. A `Friends` edge table would hold all edges that connect one person to another. The edge tables store the `Node ID` of the nodes, where every edge originates and terminates. In addition, the node tables have information about the `Object ID` corresponding to each node.

Implementing a FIFO stack

Now let's see how we can implement a stack data structure in SQL Server. The following example is a FIFO stack to emulate order numbers in a restaurant.

First, we need a table that will serve as our stack. We'll call it `OrderLog`. The first order in the order log at one end of the kitchen line will also be the first order at the other end.

```
CREATE TABLE [dbo].[OrdeLog] (
    [OrderId] [int] NOT NULL ,
    [TableNumber] [char] (10) NOT NULL

)

GO



ALTER TABLE [dbo].[OrderLog] WITH NOCHECK ADD
    CONSTRAINT [PK_OrderLog] PRIMARY KEY  CLUSTERED
    (
        [OrderId]
    )

GO
```

Creating the table structure is the first step.

The rows in the table would look something like this:

```
OrderId    TableNumber

1             A

2             B

3             A

4             C

5             D
```

Example of Orders in the OrderLog

Next, we need the two key operations (push/pop) for a stack data structure, which we'll implement as stored procedures.

```
CREATE PROCEDURE dbo.FIFO_Push (@TableNumber CHAR (10) )

AS

DECLARE @OrderId INTEGER



SELECT TOP 1 @OrderId = OrderId
FROM OrderLog
   ORDER BY OrderId
DESC

INSERT INTO OrderLog( OrderId,
TableNumber ) VALUES( @OrderId
+ 1,
 @TableNumber
)

SELECT * FROM OrderLog WHERE OrderId
= @OrderId + 1

GO



CREATE PROCEDURE dbo.FIFO_Pop

AS

DECLARE @OrderId INTEGER



SELECT TOP 1 @OrderId = OrderId
FROM OrderLog
   ORDER BY OrderId


SELECT * FROM OrderLog WHERE @OrderId
= OrderId

DELETE FROM OrderLog WHERE @OrderId
= OrderId

GO
```

If you are planning to use this code in production you can eliminate the Select statements in each procedure.

Since we have a few sample rows, you can now push and pop at will.

Implementing a LIFO stack

Implementing a LIFO stack is just as simple. The push procedure is identical to the code above. The only change is to add DESC to the line that begins "SELECT TOP 1." in the pop procedure.

Conclusion

The usage of data structures in any data processing system can give highly efficient results. There are several types of data structures that you can choose from to fit your use case scenario.

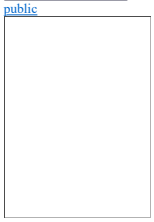
SQL has inbuilt methods and functions that implement data structures in a user-friendly style. It's clear as day that proper usage of data structures in an SQL database management system can optimize data processing to a significant degree.

- Tags:
- SQL

Spread the word

- Share
- Tweet
- Share
- Copy
- Email

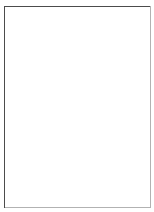
https://arctype.com/blo
public



Next article

Building a REST API with Node.js, MySQL, and Express

public



Previous article

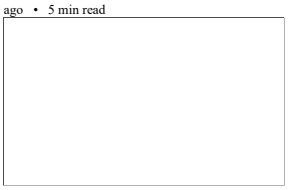
How Covid changed the way we work with SQL

Keep reading

public 

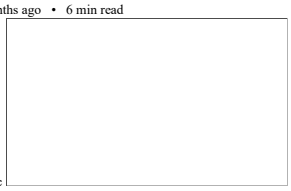
Factors to consider when choosing a SQL client

3 days ago • 5 min read

public 

Query S3 with SQL using S3 Select

2 months ago • 6 min read

public 

10 AWS Services that use SQL

2 months ago • 10 min read

Main links

- Changelog
- Postgres
- MySQL
- SQLite

Secondary links

- Figma ERD Template
- Database Security Checklist
- Write for Us
- Blog Home

Social links

- Twitter

Published with Ghost