SQL Server table structure overview

March 7, 2018 by Ahmad Yaseen

Microsoft SQL Server is a relational database management systems (RDBMS) that, at its fundamental level, stores the data in tables. The tables are the database objects that behave as containers for the data, in which the data will be logically organized in rows and columns format. Each row is considered as an entity that is described by the columns that hold the attributes of the entity. For example, the customers table contains one row for each customer, and each customer is described by the table columns that hold the customer information, such as the CustomerName and CustomerAddress. The table rows have no predefined order, so that, to display the data in a specific order, you would need to specify the order that the rows will be returned in. Tables can be also used as a security boundary/mechanism, where database users can be granted permissions at the table level.

Table basics

SQL Server tables are contained within database object containers that are called Schemas. The schema also works as a security boundary, where you can limit database user permissions to be on a specific schema level only. You can imagine the schema as a folder that contains a list of files. You can create up to 2,147,483,647 tables in a database, with up to 1024 columns in each table. When you design a database table, the properties that are assigned to the table and the columns within the table will control the allowed data types and data ranges that the table accepts. Proper table design, will make it easier and faster to store data into and retrieve data from the table.

Special table types

In addition to the basic user defined table, SQL Server provides us with the ability to work with other special types of tables. The first type is the **Temporary Table** that is stored in the tempdb system database. There are two types of temporary tables: a local temporary table that has the single number sign prefix (#) and can be accessed by the current connection only, and the Global temporary table that has two number signs prefix (##) and can be accessed by any connection once created.

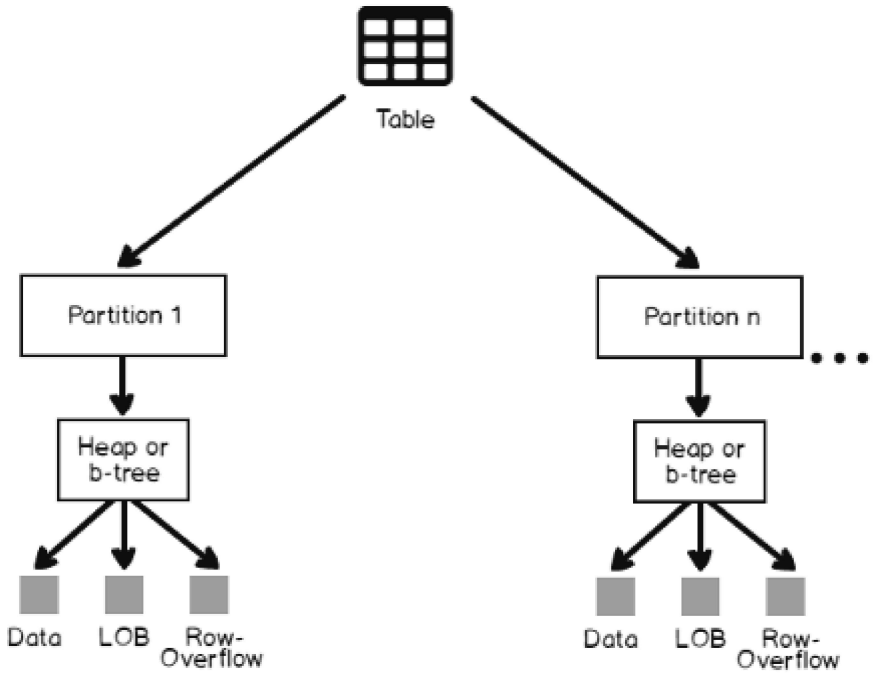
A **Wide Table** is a table the uses the Sparse Column to for optimized storage for the NULL values, reducing the space consumed by the table and increasing the number of columns allowed in that table to 30K columns.

System Tables are a special type of table in which the SQL Server Engine stores information about the SQL Server instance configurations and objects information, that can be queried using the system views.

Partitioned Tables are tables in which the data will be divided horizontally into separate unites in the same filegroup or different filegroups, based on a specific key, to enhance the data retrieval performance.

Physical implementation

Physically, SQL Server tables are stored in ta database as a set of 8 KB pages. Table pages are stored by default in a single partition that resides in the PRIMARY default filegroup. A table can be also stored in multiple partitions, in which each group of rows will be stored in a specific partition, in one or more filegroups, based on a specific column. Each table partition contains data rows in a heap or clustered index structure, that are managed in allocation units, depending on the data types of each column in the data rows.The image based on Microsoft the SQL Server Book Online article Table and Index Organization summarizes the table structure:

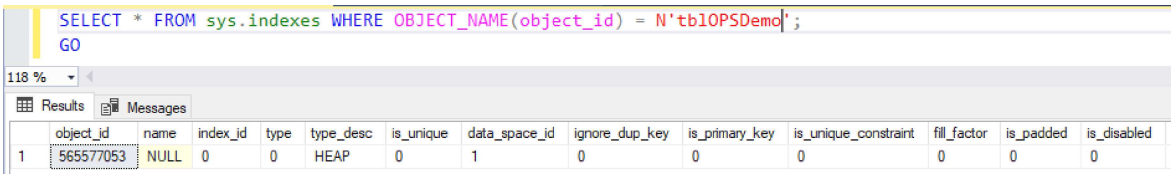


As you can see from the previous image, the data pages for the SQL Server table can be organized within each partition in two ways: in Heap or B-Tree Clustered tables. In the **Heap** table, the data rows are not stored in any particular order within each data page. In addition, there is no particular order to control the data pages sequence, that is not linked in a linked list. This is due to the fact that the heap table contains no clustered index. As there is no enforced order for the rows in the heap table, the data rows will be added to the first available location within the table's pages, after checking that it has sufficient space. If no space is available, additional pages will be added to the

table and the rows will be inserted into these new pages. This is why the data order cannot be predicted. Only the order of the returned rows can be enforced using the ORDER BY clause in the SELECT statement.

Heap table

When you store data in a heap table, the rows in that table are identified by a reference to the identifier of that row (RID) that contains the file number, the data page number and the slot of that data page. The heap table has one row in the sys.partitions system object per each partition with index_id value equal to 0. You can query the sys.indexes system object also to show the heap table index details, that will show you that, the id of that index is 0 and the type of it is HEAP, as shown below:



```
SELECT * FROM sys.indexes WHERE OBJECT_NAME(object_id) = N'tblOPSDemo';
GO
```

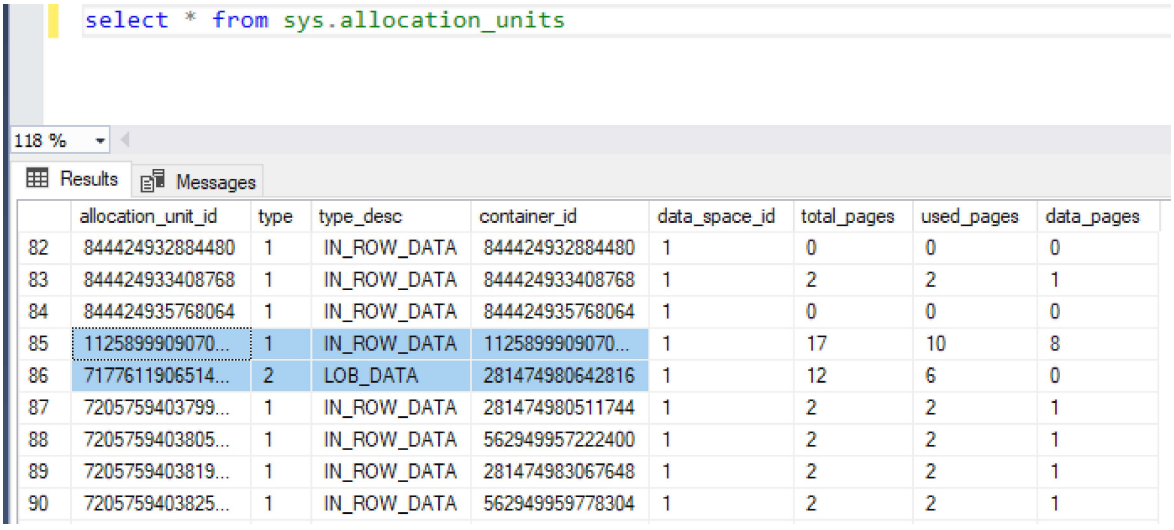
118 %

Results | Messages

	object_id	name	index_id	type	type_desc	is_unique	data_space_id	ignore_dup_key	is_primary_key	is_unique_constraint	fill_factor	is_padded	is_disabled
1	565577053	NULL	0	0	HEAP	0	1	0	0	0	0	0	0

Each partition in the heap table will have a heap structure with the data allocation units to store and manage the data in that partition depends on the data types in the heap. For example, all heaps will contain IN_ROW_DATA allocation unit and may contain the LOB_DATA allocation unit if it contains large object data or ROW_OVERFLOW_DATA allocation unit if it contains variable length columns that exceed the row size limit of 8K bytes.

Although the heap has no index structure that manages the pages and the data allocation, SQL Server Engine uses an **Index Allocation Map** (IAM) to keep an entry for each page to track the allocation of these available pages. The IAM is considered as the only logical connection between the data pages, that the SQL Server Engine will use to move through the heap. The sys.allocation_units system object can be used to list all allocation units in a specific database, as shown below:



```
select * from sys.allocation_units
```

118 %

Results | Messages

	allocation_unit_id	type	type_desc	container_id	data_space_id	total_pages	used_pages	data_pages
82	844424932884480	1	IN_ROW_DATA	844424932884480	1	0	0	0
83	844424933408768	1	IN_ROW_DATA	844424933408768	1	2	2	1
84	844424935768064	1	IN_ROW_DATA	844424935768064	1	0	0	0
85	1125899909070...	1	IN_ROW_DATA	1125899909070...	1	17	10	8
86	7177611906514...	2	LOB_DATA	281474980642816	1	12	6	0
87	7205759403799...	1	IN_ROW_DATA	281474980511744	1	2	2	1
88	7205759403805...	1	IN_ROW_DATA	562949957222400	1	2	2	1
89	7205759403819...	1	IN_ROW_DATA	281474983067648	1	2	2	1
90	7205759403825...	1	IN_ROW_DATA	562949959778304	1	2	2	1

Additional information about the first IAM page, the first page, and the root page can be viewed by querying the sys.system_internals_allocation_units system object, as shown below:

```
SELECT * FROM sys.system_internals_allocation_units
GO
```

118 %

Results | Messages

	allocation_unit_id	type	type_desc	container_id	filegroup_id	total_pages	used_pages	data_pages	first_page	root_page	first_iam_page
82	844424932884480	1	IN_ROW_DATA	844424932884480	1	0	0	0	0x000000000000	0x000000000000	0x000000000000
83	844424933408768	1	IN_ROW_DATA	844424933408768	1	2	2	1	0x4F0000000100	0x4F0000000100	0x500000000100
84	844424935768064	1	IN_ROW_DATA	844424935768064	1	0	0	0	0x000000000000	0x000000000000	0x000000000000
85	1125899909070...	1	IN_ROW_DATA	1125899909070...	1	17	10	8	0x7A0000000100	0xD60000000100	0x7B0000000100
86	7177611906514...	2	LOB_DATA	281474980642816	1	12	6	0	0x8E0000000100	0x8E0000000100	0x2D0000000100
87	7205759403799...	1	IN_ROW_DATA	281474980511744	1	2	2	1	0x9F0000000100	0x300000000100	0x0D0000000100
88	7205759403805...	1	IN_ROW_DATA	562949957222400	1	2	2	1	0x240000000100	0x240000000100	0x540000000100
89	7205759403819...	1	IN_ROW_DATA	281474983067648	1	2	2	1	0x6E0000000100	0x6E0000000100	0x720000000100
90	7205759403825...	1	IN_ROW_DATA	562949959778304	1	2	2	1	0x730000000100	0x730000000100	0x860000000100

To perform a table scan on the heap table, SQL Server Engine will scan the IAM pages serially to locate the extents that are holding the requested data. Recall that the extent consists of 8 pages. SQL Server uses the first_iam_page value, that points to the first IAM page in the chain of IAM pages, shown in the previous snapshot to locate the IAM page that contains the allocation address of the heap table, where SQL Server will use that address in the IAM to find the requested heap data pages.

When a data modification operation is performed on the heap table data pages, **Forwarding Pointers** will be inserted into the heap to point to the new location of the moved data. These forwarding pointers will cause performance issues over time due to visiting the old/original location vs the new location specified by the forwarding pointers to get a specific value. Starting from SQL Server version 2008, a new method was introduced to overcome the forwarding pointers performance issue, by using the ALTER TABLE REBUILD command, that will rebuild the heap table, as shown below:

```
ALTER TABLE tblOPSDemo REBUILD
```

118 %

Messages

```
Commands completed successfully.
```

It is better not to keep the table, with no sorting mechanism, when you have large tables that you use to retrieve data from in specific sorting or grouping order as that will result in very bad performance. To avoid such performance issues, the table can be designed with internal ordering logic. This can be achieved by converting the table from heap table to a clustered table.

Clustered table

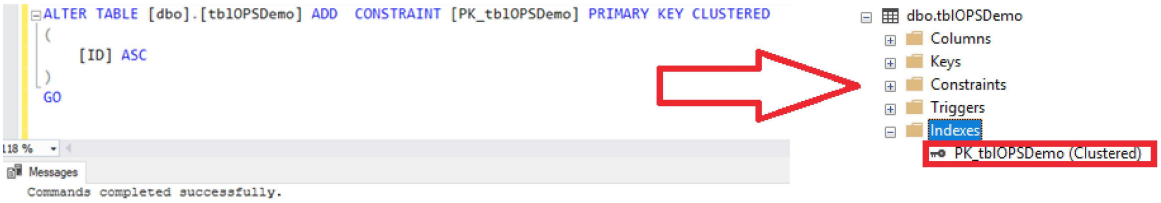
A clustered table is a table that has a predefined clustered index on one column or multiple columns of the table that defines the storing order of the rows within the data pages and the order of the pages within the table, based on the clustered index key. As the table rows can be stored only in single order, you can define only one clustered index on each table.

It is a common mistake to assume that the clustered index pages are physically sorted based on the clustered index key. SQL Server always tries to align between the physical and logical order while creating the index, but once the data is deleted or modified, this order will be broken,

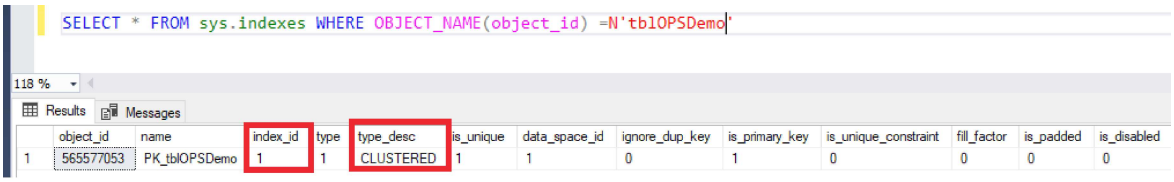
leading to the common issue of fragmentation. When ana INSERT operation is performed on the clustered table, SQL Server will locate it in the correct logical position, if there is a suitable space for it, otherwise, the page will be split into two pages to fit the newly inserted data.

A clustered index is built using the B-tree structure, with one B-tree per each partition of the clustered table, in which the data pages in each level of the clustered index, from the root level until the leaf level, are linked in a doubly-linked list. This provides for fast data navigation due to theretrieval process, based on the clustered index key values. Similar to the heap structure, each B-tree will contain IN_ROW_DATA allocation unit and may contain the LOB_DATA allocation unit if it contains large object data or ROW_OVERFLOW_DATA allocation unit if it contains variable length columns that exceed the row size limit of 8K bytes.

Let us create a primary key constraint on the previous heap table, that will add a clustered index automatically to that table as shown below:



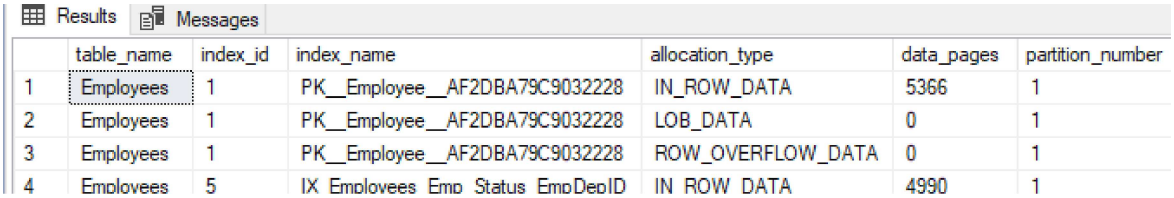
Querying the sys.indexes system object for that table again, you will see that the ID of the clustered index is 1 as shown in the index details below:



We can also get detailed information about all available allocation units in one of our large tables, the Employee table for example, by querying the sys.allocation_units system object and join it with the sys.partitions, sys.objects and sys.indexes system views, using the T-SQL statement below:

```
SELECT Obj.name AS table_name,Par.index_id, IDX.name AS index_name , AllUn.type
_desc AS allocation_type, AllUn.data_pages, partition_number
FROM sys.allocation_units AS AllUn
    JOIN sys.partitions AS Par ON AllUn.container_id = Par.partition_id
    JOIN sys.objects AS Obj ON Par.object_id = Obj.object_id
    JOIN sys.indexes AS IDX ON Par.index_id = IDX.index_id AND IDX.object_id =
Par.object_id
WHERE Obj.name = N'Employees'
```

The result will show us a list of all partitions that shape the Employee table, with all available data allocation types on each partition, and the number of data pages on each allocation unit, as shown below:



5	Employees	5	IX_Employees_Emp_Status_EmpDepID	LOB_DATA	0	1
6	Employees	5	IX_Employees_Emp_Status_EmpDepID	ROW_OVERFLOW_DATA	0	1

Conclusion

In this article, we described, in detail, the structure of the SQL Server main data storage unit, the table. We mentioned also the different types of user-defined tables that can be used to store your data. After that, we went through the differences between heap tables and clustered tables from different aspects, how to covert the tables between these two types, as well as how to get statistical information about the heap and clustered tables. In the next article, we will go through the main concepts of the SQL Server indexes. Keep tuned.

Table of contents

Ahmad Yaseen

Ahmad Yaseen is a Microsoft Big Data engineer with deep knowledge and experience in SQL BI, SQL Server Database Administration and Development fields.

He is a Microsoft Certified Solution Expert in Data Management and Analytics, Microsoft Certified Solution Associate in SQL Database Administration and Development, Azure Developer Associate and Microsoft Certified Trainer.

Also, he is contributing with his SQL tips in many blogs.

View all posts by Ahmad Yaseen

Related Posts:

1. **Index Strategies – Part 1 – Choosing the right table structure**
2. **SQL Server index structure and concepts**
3. **An overview of the SQL table variable**
4. **Forwarded Records Performance issue in SQL Server**
5. **SQL query performance killers – understanding poor database indexing**

Indexes

84,295 Views