The ultimate in-depth reference

Hundreds of timesaving solutions

Supremely well-organized, packed with expert advice

Microsoft

SQL Server 2022 Administration Inside OUT

Randolph West • William Assaf • Elizabeth Noble
Meagan Longoria • Joey D'Antoni • Louis Davidson

With contributions from: William Carter, Josh Smith, Melody Zacharias

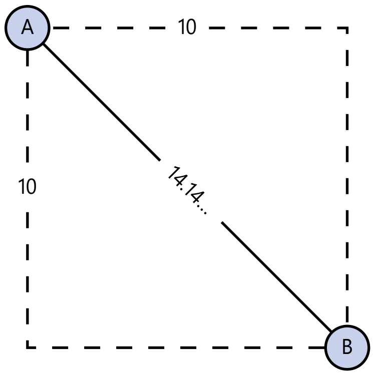

Figure 7-1 The geometry defined in the sample script.

Using spatial data types in a database is valuable when you use the Database Engine to perform spatial queries. You have probably experienced the results of spatial queries in many applications—for example, when searching for nearby pizza restaurants on Bing Maps. Application code can certainly also perform those spatial queries; however, it would require the database to return all pizza restaurants along with their coordinates. By performing the spatial query in the database, the size of the data returned to the application is significantly reduced. SQL Server supports indexing spatial data such that spatial queries can perform optimally.

➤ **For a complete reference on the geometry and geography data types, the methods they support, and spatial reference identifiers, visit *https://learn.microsoft.com/sql/t-sql/spatial-geometry/spatial-types-geometry-transact-sql* and *https://learn.microsoft.com/sql/t-sql/spatial-geography/spatial-types-geography*.**

NOTE

For an example of the `geography` data type, refer to the WideWorldImporters sample database. The `Application.StateProvinces` table includes a `Border` column of type `geography`. To visually see the `geography` data type at work, run a `SELECT` statement on all rows in the table using SQL Server Management Studio (SSMS). In addition to the row results, SSMS will display a Spatial results tab on which a map of the globe will be drawn.

The XML data type

The `xml` data type is designed to store XML documents or snippets. But support for XML goes beyond just storing XML data. The `XML` data type can enforce an XML schema, in which case the column is referred to as *typed*. XML data can also be queried using XQuery syntax. SQL Server further supports XML by formatting relational data output as XML or retrieving XML data in a relational structure.

A relational database is generally used to store highly structured data, by which we mean data that has a known schema. And even though schemas can change, at any given time every row in a table will have the same columns. Yet, for some scenarios, this strict schema is not appropriate. It might be necessary to accommodate storing data where different rows have different attributes. Sometimes, you can meet this requirement by adding additional nullable sparse columns.

A column set is a feature by which you can manage a group of sparse columns as XML data. Column sets come with significant limitations. Defining many sparse columns becomes onerous because a substantial number of columns can introduce challenges in working with the table. There, just storing the data as plain XML in an `xml` data type can alleviate the column sprawl. Additionally, if data is frequently used in XML format, it might be more efficient to store the data in that format in the database.

➤ **You can read more about sparse columns in the "Sparse columns" section later in this chapter. For detailed guidance on the use of column sets, see *https://learn.microsoft.com/sql/relational-databases/tables/use-column-sets*.**

Although XML data could be stored in `(n)varchar` columns, using the specialized data type allows SQL Server to provide functionality for validating, querying, indexing, and modifying the XML data.

NOTE

SQL Server 2022 introduces XML compression, which can dramatically reduce the amount of storage required for XML data and XML indexes.

➤ **Refer to the section "Large value data" earlier in this chapter for details on how `xml` data is stored and access can be optimized.**

Inside OUT

How can you work with JSON in SQL Server just like XML?

SQL Server 2016 introduced support for JSON, though it is not a distinct data type like XML. JSON support includes parsing, querying, modifying, and transforming JSON stored in varchar columns using functions. The brief sample that follows illustrates how to check if a value is valid JSON, using the `ISJSON()` function, and extracting a scalar value, using the `JSON_VALUE()` function.

```
DECLARE @SomeJSON nvarchar(50) = '{ "test": "passed" }';
SELECT ISJSON(@SomeJSON) IsValid, JSON_VALUE(@SomeJSON, '$.test') [Status];
```

The output from this code has two columns. The first column has the value 1 because the variable holds valid JSON data, and the second contains the value passed.

For an additional example, examine the columns `CustomFields` and `OtherLanguages` in the `Application.People` table in the WideWorldImporters sample database. `OtherLanguages` is a computed column, which extracts some JSON data from the `CustomFields` column using the `JSON_QUERY()` function.

➤ **For complete information on handling JSON-formatted data in SQL Server and Azure SQL Database, refer to *https://learn.microsoft.com/sql/relational-databases/json/json-data-sql-server*.**

The rowversion data type

This data type generates a database-wide unique binary value upon each modification of row data. This binary value increments with each `INSERT` or `UPDATE` statement that affects the row, even if no other row data is modified. A common function of this data type is as a row change indicator for use with applications that use optimistic concurrency or as a database-wide change indicator.

NOTE

The `rowversion` data type was previously known as `timestamp`. `rowversion` is the recommended name to use; `timestamp` is deprecated. Unfortunately, SSMS does not support the use of `rowversion` in the table designer or when scripting a table; it continues to use `timestamp`.

The name `timestamp` is the same as the SQL ISO standard timestamp, but it does not work according to the ISO standard. Contrary to what the `timestamp` name might imply, the data in a `rowversion` column does not map to a moment in time.

When designing tables with `rowversion`, keep the following restrictions in mind:

- A table can have only a single rowversion column. Considering the context of rowversion, this restriction is perfectly sensible, and we've included it here only for completeness.
- You cannot specify a value for the rowversion column in `INSERT` or `UPDATE` statements. However, unlike with identity or computed columns, you must specify a column list in `INSERT` statements for tables with a rowversion column. Note that specifying the column list is recommended anyway.
- Although the Database Engine will not generate duplicate rowversion values within a database, rowversion values are not unique across databases or instances.

Duplicate rowversion values can exist in a single database if a new table is created by using the `SELECT INTO` syntax. The new table's rowversion values will be the same as those of the source table. This behavior might be desired when, for example, modifying a table's schema by creating a new table and copying all the data into it. In other instances, this behavior might not be desired. In those cases, you should not include the rowversion column in the `SELECT INTO` statement. You should then alter the new table to add a rowversion column. This behavior and workaround are illustrated in an extra sample script file in the accompanying downloads for this book.

Implement optimistic concurrency

Including a rowversion column in a table is an excellent way to implement a row change indicator to achieve *optimistic concurrency*. With optimistic concurrency, a client reads data with the intent of updating it. Unlike with *pessimistic concurrency*, however, a lock is not maintained. Instead, in the same transaction as the `UPDATE` statement, the client will verify that the rowversion was not changed by another process. If it wasn't, the update proceeds. But if the rowversion no longer matches what the client originally read, the update will fail. The client application can then retrieve the current values and present the user with a notification and suitable options, depending on the application needs. Many object-relational mappers (ORMs), including Entity Framework, support using a `rowversion` column type to implement optimistic concurrency.

Inside OUT

Can you implement optimistic concurrency without rowversion?

There are other ways to implement optimistic concurrency. A client application can track the value of each individual column in the updated row, then verify that only the columns affected by its own `UPDATE` statement have not been modified. Specifically, client A reads a row of data and intends to change only the Name column. Client B reads the same row of data and updates the Address column. When client A attempts to update the Name, it finds that the Name column's value is unchanged and will proceed with the update.

This approach is suitable in some scenarios, but it has some drawbacks. First, each client needs to maintain additional state information—namely, the original value of each column. In a web application, the amount of state information to maintain can grow very large and consume a lot of memory. In a web farm scenario, maintaining such state information might require shared state configuration because the web client might not communicate with the same web server on the `POST` that it did on the `GET`.

Perhaps more importantly, the data row can be inconsistent after the second update. If each client updates a column in the same row, the row's data might not reflect a valid business scenario. Certainly, the row's values would not reflect what each client believes it would be.

Alternatively, you can use read committed snapshot isolation (RCSI) to enable optimistic concurrency at the database level, which is discussed in more detail in Chapter 6, "Provision and configure SQL Server databases."

The uniqueidentifier data type

The `uniqueidentifier` data type stores a 16-byte value known as a *globally unique identifier* (*GUID*). SQL Server can generate GUIDs using one of two functions: `NEWID()` and `NEWSEQUENTIALID()`. `NEWSEQUENTIALID()` generates a GUID that is greater than a previously generated GUID by this function *since the last restart of the server*. You can use `NEWSEQUENTIALID()` only in a default constraint for a column; it is more suitable for use as a clustered primary key than `NEWID()`. Unlike `NEWID()`, which generates random values, the increasing nature of the GUIDs generated by `NEWSEQUENTIALID()` means that data and index pages will fill completely.

NOTE

Although GUIDs generated using `NEWID()` were originally generated by incorporating a system's network interface card (NIC) MAC address, this has not been the case for many years, because it might have been possible to identify the system on which it was created. All GUIDs in SQL Server use a pseudorandom value, according to the UUID version 4 standard. The chance of collision is extremely low.

However, the `NEWSEQUENTIALID()` function *is* dependent on the MAC address of the machine's network interface. This means that the starting point of the sequence generated by `NEWSEQUENTIALID()` could change when the machine's network interface changes. A NIC change can occur with regularity on virtualized and PaaS platforms. With `NEWSEQUENTIALID()`, you will eventually experience fragmentation because the sequential GUIDs will have a smaller value than the previous sequence after a restart.

➤ **The `uniqueidentifier` data type plays an important role in some replication techniques. For more information, see Chapter 11, "Implement high availability and disaster recovery."**

The hierarchyid data type

The `hierarchyid` data type enables an application to store and query hierarchical data in a tree structure. A tree structure means that a row will have zero or one parent and zero or more children. There is a single root element denoted by a single forward slash (/). `hierarchyid` values are stored as a binary format but are commonly represented in their string format. Each element at the same level in the hierarchy (referred to as a *sibling*) has a unique numeric value (which might include a decimal point). In the string representation of a `hierarchyid` value, each level is separated by a forward slash. The string representation always begins with a slash (to denote the root element) and ends with a slash.

For example, as illustrated in Figure 7-2, a `hierarchyid` whose string representation is /1/10/ is a descendant of the /1/ element, which itself is a descendant of the implicit root element /. It must be noted, however, that SQL Server does not enforce the existence of a row with the ancestor element. This means it is possible to create an element /3/1/ without its ancestor /3/ being a value in a row. Implicitly, it is a child of /3/, even if no row with `hierarchyid` value /3/ exists. Similarly, the row with `hierarchyid` element /1/ can be deleted if another row has `hierarchyid` value /1/10/. If you don't want this, the application or database will need to include logic to enforce the existence of an ancestor when inserting and to prevent the deletion of an ancestor.

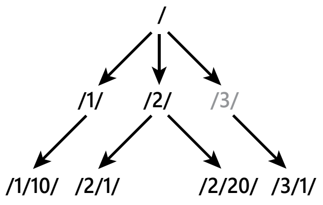

Figure 7-2 `hierarchyid` values. The value /3/ is in gray to indicate it is implicit.

Perhaps surprisingly, SQL Server does not enforce uniqueness of the `hierarchyid` values unless you define a unique index or constraint on the `hierarchyid` column. It is, therefore, possible for the /3/1/ element to be defined twice. This is likely not the desired situation, so we recommend that you ensure uniqueness of the `hierarchyid` values.

Using the `hierarchyid` data type is appropriate if the tree is most commonly queried to find descendants, such as children, children-of-children, and more. This is because `hierarchyid` processes rows depth-first if it is indexed. You can create a breadth-first index by adding a computed column to the table (which uses the `.GetLevel()` method on the `hierarchyid` column) and then creating an index on the computed column followed by the `hierarchyid` column. You cannot, however, use a computed column in a clustered index, so this solution will still be less efficient compared to creating a clustered index on the `hierarchyid` value alone.

A `hierarchyid` method worth mentioning is `.GetAncestor()`. This method returns the `hierarchyid` value of the current node's parent. Conversely, `.IsDescendantOf()` determines whether a node is a descendant, direct or otherwise, of the `hierarchyid` provided as the function's parameter.

➤ **For a complete overview of the `hierarchyid` data type, refer to *https://learn.microsoft.com/sql/relational-databases/hierarchical-data-sql-server*.**

The sql_variant data type

The `sql_variant` data type enables a single column to store data of diverse types. You can also use this type as a parameter or a variable. In addition to storing the actual value, each `sql_variant` instance also stores metadata about the value, including its system data type, maximum size, scale and precision, and collation. Using `sql_variant` can be indicative of a poor database design, and you should use it judiciously. Client libraries that do not know how to handle that data might convert it to `nvarchar(4000)`, with potential consequences for data that doesn't convert well to character data.