**Describe normalization**

Database normalization is a design process used to organize a given set of data into tables and columns in a database. Each table should contain data relating to a specific ‘thing’ and only have data that supports that same ‘thing’ included in the table. The goal of this process is to reduce duplicate data contained within your database, to reduce performance degradation of database inserts and updates. For example, a customer address change is much easier to implement if the only place the customer address is stored is in the Customers table. The most common forms of normalization are first, second, and third normal form and are described below.

**First normal form**

First normal form has the following specifications:

* Create a separate table for each set of related data
* Eliminate repeating groups in individual tables
* Identify each set of related data with a primary key

In this model, you shouldn't use multiple columns in a single table to store similar data. For example, if product can come in multiple colors, you shouldn't have multiple columns in a single row containing the different color values. The first table, below (ProductColors), isn't in first normal form as there are repeating values for color. For products with only one color, there's wasted space. And what if a product came in more than three colors? Rather than having to set a maximum number of colors, we can recreate the table as shown in the second table, ProductColor. We also have a requirement for first normal form that there's a unique key for the table, which is column (or columns) whose value uniquely identifies the row. Neither of the columns in the second table is unique, but together, the combination of ProductID and Color is unique. When multiple columns are needed, we call that a composite key.

|  |  |  |  |
| --- | --- | --- | --- |
| ProductID | Color1 | Color2 | Color3 |
| 1 | Red | Green | Yellow |
| 2 | Yellow |  |  |
| 3 | Blue | Red |  |
| 4 | Blue |  |  |
| 5 | Red |  |  |

|  |  |
| --- | --- |
| ProductID | Color |
| 1 | Red |
| 1 | Green |
| 1 | Yellow |
| 2 | Yellow |
| 3 | Blue |
| 3 | Red |
| 4 | Blue |
| 5 | Red |

The third table, ProductInfo, is in first normal form because each row refers to a particular product, there are no repeating groups, and we have the column ProductID to use as a Primary Key.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ProductID | ProductName | Price | ProductionCountry | ShortLocation |
| 1 | Widget | 15.95 | United States | US |
| 2 | Foop | 41.95 | United Kingdom | UK |
| 3 | Glombit | 49.95 | United Kingdom | UK |
| 4 | Sorfin | 99.99 | Republic of the Philippines | RepPhil |
| 5 | Stem Bolt | 29.95 | United States | US |

**Second normal form**

Second normal form has the following specification, in addition to those required by first normal form:

* If the table has a composite key, all attributes must depend on the complete key and not just part of it.

Second normal form is only relevant to tables with composite keys, like in the table ProductColor, which is the second table above. Consider the case where the ProductColor table also includes the product’s price. This table has a composite key on ProductID and Color, because only using both column values can we uniquely identify a row. If a product’s price doesn't change with the color, we might see data as shown in this table:

|  |  |  |
| --- | --- | --- |
| ProductID | Color | Price |
| 1 | Red | 15.95 |
| 1 | Green | 15.95 |
| 1 | Yellow | 15.95 |
| 2 | Yellow | 41.95 |
| 3 | Blue | 49.95 |
| 3 | Red | 49.95 |
| 4 | Blue | 99.95 |
| 5 | Red | 29.95 |

The table above is **not** in second normal form. The price value is dependent on the ProductID but not on the Color. There are three rows for ProductID 1, so the price for that product is repeated three times. The problem with violating second normal form is that if we have to update the price, we have to make sure we update it everywhere. If we update the price in the first row, but not the second or third, we would have something called an ‘update anomaly’. After the update, we wouldn’t be able to tell what the actual price for ProductID 1 was. The solution is to move the Price column to a table that has ProductID as a single column key, because that is the only column that Price depends on. For example, we could use Table 3 to store the Price.

If the price for a product was different based on its color, the fourth table would be in the second normal form, since the price would depend on both parts of the key: the ProductID and the Color.

**Third normal form**

Third normal form is typically the aim for most OLTP databases. Third normal form has the following specification, in addition to those required by second normal form:

* All nonkey columns are non-transitively dependent on the primary key.

The transitive relationship implies that one column in a table is related to other columns, through a second column. Dependency means that a column can derive its value from another, as a result of a dependency. For example, your age can be determined from your date of birth, making your age dependent on your date of birth. Refer back to the third table, ProductInfo. This table is in second normal form, but not in third. The *ShortLocation* column is dependent on the ProductionCountry column, which isn't the key. Like second normal form, violating third normal form can lead to update anomalies. We would end up with inconsistent data if we updated the *ShortLocation* in one row but didn't update it in all the rows where that location occurred. To prevent this, we could create a separate table to store country/region names and their shortened forms.

**Denormalization**

While the third normal form is theoretically desirable, it isn't always possible for all data. In addition, a normalized database doesn't always give you the best performance. Normalized data frequently requires multiple join operations to get all the necessary data returned in a single query. There's a tradeoff between normalizing data when the number of joins required to return query results has high CPU utilization, and denormalized data that has fewer joins and less CPU required, but opens up the possibility of update anomalies.

**Note**

Denormalized data is not the same as unnormalized. For denormalization, we start by designing tables that are normalized. Then we can add additional columns to some tables to reduce the number of joins required, but as we do so, we are aware of the possible update anomalies. We then make sure we have triggers or other kinds of processing that will make sure that when we perform an update, all the duplicate data is also updated.

Denormalized data can be more efficient to query, especially for read heavy workloads like a data warehouse. In those cases, having extra columns may offer better query patterns and/or more simplistic queries.

**Star schema**

While most normalization is aimed at OLTP workloads, data warehouses have their own modeling structure, which is usually a **denormalized** model. This design uses fact tables, which record measurements or metrics for specific events like a sale, and joins them to dimension tables, which are smaller in terms of row count, but may have a large number of columns to describe the fact data. Some example dimensions would include inventory, time, and/or geography. This design pattern is used to make the database easier to query and offer performance gains for read workloads.

The above image shows an example of a star schema, including a *FactResellerSales* fact table, and dimensions for date, currency, and products. The fact table contains data related to the sales transactions, and the dimensions only contain data related to a specific element of the sales data. For example, the *FactResellerSales* table contains only a *ProductKey* to indicate which product was sold. All of the details about each product are stored in the *DimProduct* table, and related back to the fact table with the *ProductKey* column.

Related to star schema design is a snowflake schema, which uses a set of more normalized tables for a single business entity. The following image shows an example of a single dimension for a snowflake schema. The Products dimension is normalized and stored in three tables called *DimProductCategory*, *DimProductSubcategory*, and *DimProduct*.

The main difference between star and snowflake schemas is that the dimensions in a snowflake schema are normalized to reduce redundancy, which saves storage space. The tradeoff is that your queries require more joins, which can increase your complexity and decrease performance.

**Choose appropriate data types**

Completed100 XP

* 4 minutes

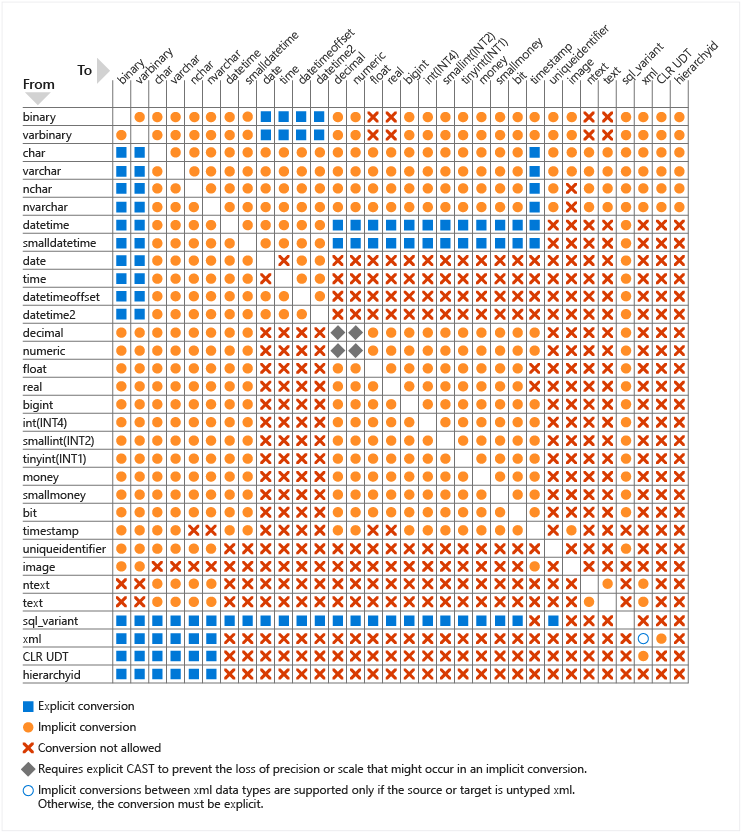
SQL Server offers a wide variety of data types to choose from, and your choice can affect performance in many ways. While SQL Server can convert some data types automatically (we call this an ‘implicit conversion’), conversion can be costly and can also negatively affect query plans. The alternative is an explicit conversion, where you use the CAST or CONVERT function in your code to force a data type conversion.

Additionally, choosing data types that are much larger than needed can cause wasted space and require more pages than is necessary to be read. It's important to choose the right data types for a given set of data—which will reduce the total storage required for the database and improve the performance of queries executed.

**Note**

In some cases, conversions are not possible at all. For example, a date cannot be converted to a bit. Conversions can negatively impact query performance by causing index scans where seeks would have been possible, and additional CPU overhead from the conversion itself.

The image below indicates in which cases SQL Server can do an implicit conversion and in which cases you must explicitly convert data types in your code.



SQL Server offers a set of system supplied data types for all data that can be used in your tables and queries. SQL Server allows the creation of user defined data types in either T-SQL or the .NET framework.

**Design indexes**

SQL Server has several index types to support different types of workloads. At a high level, an index can be thought of as an on-disk structure that is associated with a table or a view, that enables SQL Server to more easily find the row or rows associated with the index key (which consists of one or more columns in the table or view), compared to scanning the entire table.

**Clustered indexes**

A common DBA job interview question is to ask the candidate the difference between a clustered and nonclustered index, as indexes are a fundamental data storage technology in SQL Server. A clustered index is the underlying table, stored in sorted order based on the key value. There can only be one clustered index on a given table, because the rows can be stored in one order. A table without a clustered index is called a heap, and heaps are typically only used as staging tables. An important performance design principle is to keep your clustered index key as narrow as possible. When considering the key column(s) for your clustered index, you should consider columns that are unique or that contain many distinct values. Another property of a good clustered index key is for records that are accessed sequentially, and are used frequently to sort the data retrieved from the table. Having the clustered index on the column used for sorting can prevent the cost of sorting every time that query executes, because the data will be already stored in the desired order.

**Note**

When we say that the table is ‘stored’ in a particular order, we are referring to the logical order, not necessarily the physical, on-disk order. Indexes have pointers between pages, and the pointers help create the logical order. When scanning an index ‘in order’, SQL Server follows the pointers from page to page. Immediately after creating an index, it is most likely also stored in physical order on the disk, but after you start making modifications to the data, and new pages need to be added to the index, the pointers will still give us the correct logical order, but the new pages will most like not be in physical disk order.

**Nonclustered indexes**

Nonclustered indexes are a separate structure from the data rows. A nonclustered index contains the key values defined for the index, and a pointer to the data row that contains the key value. You can add another nonkey column to the leaf level of the nonclustered index to cover more columns using the included columns feature in SQL Server. You can create multiple nonclustered indexes on a table.

An example of when you need to add an index or add columns to an existing nonclustered index is shown below:

The query plan indicates that for each row retrieved using the index seek, more data will need to be retrieved from the clustered index (the table itself). There's a nonclustered index, but it only includes the product column. If you add the other columns in the query to a nonclustered index as shown below, you can see the execution plan change to eliminate the key lookup.

The index created above is an example of a covering index, where in addition to the key column you're including extra columns to cover the query and eliminate the need to access the table itself.

Both nonclustered and clustered indexes can be defined as unique, meaning there can be no duplication of the key values. Unique indexes are automatically created when you create a PRIMARY KEY or UNIQUE constraint on a table.

The focus of this section is on b-tree indexes in SQL Server—these are also known as row store indexes. The general structure of a b-tree is shown below:

Each page in an index b-tree is a called an index node, and the top node of b-tree is called the root node. The bottom nodes in an index are called leaf nodes and the collection of leaf nodes is the leaf level.

Index design is a mixture of art and science. A narrow index with few columns in its key requires less time to update and has lower maintenance overhead; however it may not be useful for as many queries as a wider index that includes more columns. You may need to experiment with several indexing approaches based on the columns selected by your application’s queries. The query optimizer will generally choose what it considers to be the best existing index for a query; however, that doesn't mean that there isn't a better index that could be built.

Properly indexing a database is a complex task. When planning your indexes for a table, you should keep a few basic principles in mind:

* Understand the workloads of the system. A table that is used mainly for insert operations will benefit far less from extra indexes than a table used for data warehouse operations that are 90% read activity.
* Understand what queries are run most frequently, and optimize your indexes around those queries.
* Understand the data types of the columns used in your queries. Indexes are ideal for integer data types, or unique or non-null columns.
* Create nonclustered indexes on columns that are frequently used in predicates and join clauses, and keep those indexes as narrow as possible to avoid overhead.
* Understand your data size/volume – A table scan on a small table will be a relatively cheap operation and SQL Server may decide to do a table scan simply because it's easy (trivial) to do. A table scan on a large table would be costly.

Another option SQL Server provides is the creation of filtered indexes. Filtered indexes are best suited to columns in large tables where a large percentage of the rows has the same value in that column. A practical example would be an employee table, as shown below, that stored the records of all employees, including ones who had left or retired.

SQLCopy

CREATE TABLE [HumanResources].[Employee](

[BusinessEntityID] [int] NOT NULL,

[NationalIDNumber] [nvarchar](15) NOT NULL,

[LoginID] [nvarchar](256) NOT NULL,

[OrganizationNode] [hierarchyid] NULL,

[OrganizationLevel] AS ([OrganizationNode].[GetLevel]()),

[JobTitle] [nvarchar](50) NOT NULL,

[BirthDate] [date] NOT NULL,

[MaritalStatus] [nchar](1) NOT NULL,

[Gender] [nchar](1) NOT NULL,

[HireDate] [date] NOT NULL,

[SalariedFlag] [bit] NOT NULL,

[VacationHours] [smallint] NOT NULL,

[SickLeaveHours] [smallint] NOT NULL,

[CurrentFlag] [bit] NOT NULL,

[rowguid] [uniqueidentifier] ROWGUIDCOL NOT NULL,

[ModifiedDate] [datetime] NOT NULL)

In this table, there's a column called CurrentFlag, which indicates if an employee is currently employed. This example uses the bit datatype, indicating only two values, one for currently employed and zero for not currently employed. A filtered index with a WHERE CurrentFlag = 1, on the *CurrentFlag* column would allow for efficient queries of current employees.

You can also create indexes on views, which can provide significant performance gains when views contain query elements like aggregations and/or table joins.

**Columnstore indexes**

Columnstore offers improved performance for queries that run large aggregation workloads. This type of index was originally targeted at data warehouses, but over time columnstore indexes have been used in many other workloads in order to help solve query performance issues on large tables. As of SQL Server 2014, there are both nonclustered and clustered columnstore indexes. Like b-tree indexes, a clustered columnstore index is the table itself stored in a special way, and nonclustered columnstore indexes are stored independently of the table. Clustered columnstore indexes inherently include all the columns in a given table. However, unlike rowstore clustered indexes, clustered columnstore indexes are NOT sorted.

Nonclustered columnstore indexes are typically used in two scenarios, the first is when a column in the table has a data type that isn't supported in a columnstore index. Most data types are supported but XML, CLR, sql\_variant, ntext, text, and image aren't supported in a columnstore index. Since a clustered columnstore always contains all the columns of the table (because it IS the table), a nonclustered is the only option. The second scenario is a filtered index—this scenario is used in an architecture called hybrid transactional analytic processing (HTAP), where data is being loaded into the underlying table, and at the same time reports are being run on the table. By filtering the index (typically on a date field), this design allows for both good insert and reporting performance.

Columnstore indexes are unique in their storage mechanism, in that each column in the index is stored independently. It offers a two-fold benefit. A query using a columnstore index only needs to scan the columns needed to satisfy the query, reducing the total IO performed, and it allows for greater compression, since data in the same column is likely to be similar in nature.

Columnstore indexes perform best on analytic queries that scan large amounts of data, like fact tables in a data warehouse. Starting with SQL Server 2016 you can augment a columnstore index with another b-tree nonclustered index, which can be helpful if some of your queries do lookups against singleton values.

Columnstore indexes also benefit from batch execution mode, which refers to processing a set of rows (typically around 900) at a time versus the database engine processing those rows one at time. Instead of loading each record independently and processing them, the query engine computes the calculation in that group of 900 records. This processing model reduces the number of CPU instructions dramatically.

SELECT SUM(Sales) FROM SalesAmount;

Batch mode can provide significant performance increase over traditional row processing. SQL Server 2019 also includes batch mode for rowstore data. While batch mode for rowstore doesn't have the same level of read performance as a columnstore index, analytical queries may see up to a 5x performance improvement.

The other benefit columnstore indexes offer to data warehouse workloads is an optimized load path for bulk insert operations of 102,400 rows or more. While 102,400 is the minimum value to load directly into the columnstore, each collection of rows, called a rowgroup, can be up to approximately 1,024,000 rows. Having fewer, but fuller, rowgroups makes your SELECT queries more efficient, because fewer row groups need to be scanned to retrieve the requested records. These loads take place in memory and are directly loaded to the index. For smaller volumes, data is written to a b-tree structure called a delta store, and asynchronously loaded into the index.

In this example, the same data is being loaded into two tables, *FactResellerSales\_CCI\_Demo* and *FactResellerSales\_Page\_Demo*. The *FactResellerSales\_CCI\_Demo* has a clustered columnstore index, and the *FactResellerSales\_Page\_Demo* has a clustered b-tree index with two columns and is page compressed. As you can see each table is loading 1,024,000 rows from the *FactResellerSalesXL\_CCI* table. When SET STATISTICS TIME is ON, SQL Server keeps track of the elapsed time of the query execution. Loading the data into the columnstore table took roughly 8 seconds, where loading into the page compressed table took nearly 20 seconds. In this example, all the rows going into the columnstore index are loaded into a single rowgroup.

If you load less than 102,400 rows of data into a columnstore index in a single operation, it's loaded in a b-tree structure known as a delta store. The database engine moves this data into the columnstore index using an asynchronous process called the tuple mover. Having open delta stores can affect the performance of your queries, because reading those records is less efficient than reading from the columnstore. You can also reorganize the index with the COMPRESS\_ALL\_ROW\_GROUPS option in order to force the delta stores to be added and compressed into the columnstore indexes.