**Normalization**

**SQL Server 2008 R2**

The logical design of the database, including the tables and the relationships between them, is the core of an optimized relational database. A good logical database design can lay the foundation for optimal database and application performance. A poor logical database design can hinder the performance of the whole system.

Normalizing a logical database design involves using formal methods to separate the data into multiple, related tables. Several narrow tables with fewer columns is characteristic of a normalized database. A few wide tables with more columns is characteristic of an non-normalized database.

Reasonable normalization frequently improves performance. When useful indexes are available, the SQL Server query optimizer is efficient at selecting rapid, efficient joins between tables.

Some of the benefits of normalization include the following:

* Faster sorting and index creation.
* A larger number of clustered indexes. For more information, see [Clustered Index Design Guidelines](http://msdn.microsoft.com/en-us/library/ms190639(v=sql.105).aspx).
* Narrower and more compact indexes.
* Fewer indexes per table. This improves the performance of the INSERT, UPDATE, and DELETE statements.
* Fewer null values and less opportunity for inconsistency. This increases database compactness.

As normalization increases, the number and complexity of joins required to retrieve data also increases. Too many complex relational joins between too many tables can hinder performance. Reasonable normalization frequently includes few regularly executed queries that use joins involving more than four tables.

Sometimes, the logical database design is already fixed and total redesign is not realistic. However, even then it might be possible to normalize a large table selectively into several smaller tables. If the database is accessed through stored procedures, this schema change could occur without affecting applications. If not, it might be possible to create a view that hides the schema change from the applications.

[Description: http://i3.msdn.microsoft.com/Areas/Global/Content/clear.gif**Achieving a Well-Designed Database**](javascript:void(0))

In relational-database design theory, normalization rules identify certain attributes that must be present or absent in a well-designed database. A complete discussion of normalization rules exceeds the scope of this topic. However, there are a few rules that can help you achieve a sound database design:

* A table should have an identifier.

The fundamental rule of database design theory is that each table should have a unique row identifier, a column or set of columns used to distinguish any single record from every other record in the table. Each table should have an ID column, and no two records can share the same ID value. The column or columns serving as the unique row identifier for a table are the primary key of the table. In the AdventureWorks2008R2 database, each table contains an identity column as the primary key column. For example, VendorID is primary key for the Purchasing.Vendor table.

* A table should store only data for a single type of entity.

Trying to store too much information in a table can hinder the efficient and reliable management of the data in the table. In the AdventureWorks2008R2 sample database, the sales order and customer information is stored in separate tables. Although you can have columns that contain information for both the sales order and the customer in a single table, this design leads to several problems. The customer information, name and address, must be added and stored redundantly for each sales order. This uses additional storage space in the database. If a customer address changes, the change must be made for each sales order. Also, if the last sales order for a customer is removed from the Sales.SalesOrderHeader table, the information for that customer is lost.

* A table should avoid nullable columns.

Tables can have columns defined to allow for null values. A null value indicates that there is no value. Although it can be useful to allow for null values in isolated cases, you should use them sparingly. This is because they require special handling that increases the complexity of data operations. If you have a table with several nullable columns and several of the rows have null values in the columns, you should consider putting these columns in another table linked to the primary table. By storing the data in two separate tables, the primary table can be simple in design and still handle the occasional need for storing this information.

* A table should not have repeating values or columns.

The table for an item in the database should not contain a list of values for a specific piece of information. For example, a product in the AdventureWorks2008R2 database might be purchased from multiple vendors. If there is a column in the Production.Product table for the name of the vendor, this creates a problem. One solution is to store the name of all vendors in the column. However, this makes it difficult to show a list of the individual vendors. Another solution is to change the structure of the table to add another column for the name of the second vendor. However, this allows for only two vendors. Additionally, another column must be added if a book has three vendors.

If you find that you have to store a list of values in a single column, or if you have multiple columns for a single piece of data, such as TelephoneNumber1, and TelephoneNumber2, you should consider putting the duplicated data in another table with a link back to the primary table. The AdventureWorks2008R2 database has a Production.Product table for product information, a Purchasing.Vendor table for vendor information, and a third table, Purchasing.ProductVendor. This third table stores only the ID values for the products and the IDs of the vendors of the products. This design allows for any number of vendors for a product without modifying the definition of the tables, and without allocating unused storage space for products with a single vendor.