ITPro Today™

BUSINESS INTELLIGENCE > MICROSOFT SQL SERVER

SQL by Design: Why You Need Database Normalization

Use this technique to improve system performance and accuracy

Michelle A. Poolet | Feb 28, 1999

Database normalization, or data normalization, is a technique to organize the contents of the tables for transactional databases and data warehouses. Normalization is part of successful database design; without normalization, database systems can be inaccurate, slow, and inefficient, and they might not produce the data you expect.

Related: When Not to Normalize Your Data

Following a successful SQL Server installation (or any database management system installation), you'll have to create a database to hold the data (see SQL Server *Books Online—BOL—*for more information about how to create a database). After you've created the database framework, you must organize the files in such a way that you can easily manage them. The primary tool to help organize the data is the table, which looks like a two-dimensional structure that encompasses rows and columns.

A database table looks like a spreadsheet. One item of data, such as a first name, a last name, a phone number, or a street address, appears in each box, or cell, at each intersection of a row and column. Database designers use the term *atomicity* to describe this organization of data into one data item in each cell.

When you normalize a database , you have four goals: arranging data into logical groupings such that each group describes a small part of the whole; minimizing the amount of duplicate data stored in a database; organizing the data such that, when you modify it, you make the change in only one place; and building a database in which you can access and manipulate the data quickly and efficiently without compromising the integrity of the data in storage.

Data normalization helps you design new databases to meet these goals or to test databases to see whether they meet the goals. Sometimes database designers refer to these goals in terms such as *data integrity, referential integrity,* or *keyed data access*. Ideally, you normalize data before you create database tables. However, you can also use these techniques to test an existing database.

Data normalization is primarily important in the transactional, or online transactional processing (OLTP), database world, in which data modifications (e.g., inserts, updates, deletes) occur rapidly and randomly throughout the stored data. In contrast, a data warehouse contains a large amount of denormalized and summarized data—precalculated to avoid the performance penalty of ad hoc joins. In a data warehouse, updates happen periodically under extremely controlled circumstances. End users' updates to data in data warehouses are uncommon. This article addresses the normalization of OLTP data.

Data and Information

Data and information are terms people use interchangeably in everyday speech, but they mean different things. Data are raw facts, such as a name, a number, or a date. Information is organized, processed data. A data item (e.g., the date 7/15/99) means little. When you associate the data with other data items, such as a deadline and a subject, you can create information. For example, the deadline for your next project might be 7/15/99. You store data in a database; you retrieve information from the database.

One cornerstone of database design and data normalization is that data organization for storage differs from the information most people want to see. For example, a manager of a sporting goods supply company might want to see, for one sale, who the customer was, the destination of the order, the billing address, the contact phone number, the placement time of the order, the order's shipping destination, when and how delivery occurred, what articles the order included, and which of the company's employees was the key person in the sale. The organization of data in the company's database differs from the particular information the manager wants. Screen 1 shows a diagram of the autogenerated Order Entry database from Microsoft Access 97. I'll use this database to illustrate the concepts in this article.

Business Rules

You need to conduct research at your company before you can begin to normalize a database. You need to perform a requirements analysis, which will identify policies and procedures and will list the business rules for them. You must have consensus on what the rules mean. By consensus, I mean that everyone who uses the database must agree on the definition and the use of these data items. Without consensus, if you ask three people in the company to define what *customer* means, you might get three different answers. To one person, a customer is the company that buys products and services. To a second person, the customer is the contact person for the company who buys product and services. To a third person, the customer is someone who might be interested in buying products and services. Some terms are standard, but under no circumstances can you assume the definition or meaning of a term. Confirm meanings of terms, and confirm how your company uses these terms.

You can use schemes, or methodologies, to guide you through the requirements-analysis phase of a database design. Think of schemes as playbooks or recipes for database design. If you are using a methodology and like it, continue to use it, no matter how small or insignificant the database design project. If you don't have a favorite methodology, you might want to explore techniques such as Bachman, IDEF1X (favored by the Department of Defense), or the new object-oriented Unified Modeling Language (UML—the scheme for the Microsoft Repository). If you're a Visual Basic (VB) programmer, you can download a free entry-level Visual Modeler from http://www.microsoft.com/vstudio . Using a scheme helps you analyze and design your database.

Relationships

After you know what tables you need, the next step in data normalization is to understand relationships and how they pertain to the data you're working with. A database is an organized, integrated collection of data items. The integration is important; data items relate to other data items, and groups of related data items—called entities—relate to other entities. The relationships between entities can be one of three types, one-to-one (1:1), one-to-many (1:M), and many-to-many (M:N).

Binary relationships, those relationships that involve two entities, are the basis for all other relationship sets. Figure 1 displays these relationships through a technique called entity relationship modeling (ERM). Entities are of two types: noun-type entities and verb-type entities. Noun-type entities represent people, places, and things. Verb-type entities represent actions and interactions between the noun-type entities.

ERM is a way to graphically represent the architecture of a database and to model the informational requirements. You can choose among many entity-modeling schemes. Personal preference, flexibility, functionality of the scheme (what it will let you do and how it will limit you) and established corporate standards will help you choose a scheme. Figure 1 uses the crow's foot scheme, or methodology. Screen 1 uses Microsoft's proprietary ERM methodology.

So many different entity modeling methodologies exist that there is not a true standard. However, one commonly accepted rule for good modeling is that entity names are always expressed in the singular. You can identify the type of relationship between singular entity names more easily than you can determine the relationship between plural entity names. A second, less common, rule is that entity names are always capitalized.

The result of your entity modeling efforts is an entity-relationship diagram (ERD). Figure 1 shows three small ERDs, each of which represents a binary relationship between two entities. A rectangle represents an entity. A line with symbols at each end joins two entities. The symbols are cardinality indicators, each of which shows how many of one entity relates to how many of another entity. The symbols differ from one methodology to another.

The one-to-one (1:1) relationship means that each instance of one entity (CUSTOMER) relates to one instance of a second entity (CREDIT_CHECK). Each CREDIT_CHECK relates back to one CUSTOMER.

The one-to-many (1:M) relationship means that each instance of one entity (ORDER) relates to one or more instances of a second entity (PAYMENT). Each PAYMENT relates back to one ORDER.

The many-to-many (M:N) relationship means that many instances of one entity (ORDER) relate to many instances of a second entity (PRODUCT). Stated a different way, an ORDER can include many PRODUCTS, and a PRODUCT can be part of many ORDERS. The M:N relationship is an interaction between two noun-type entities, and it results in the creation of a verb-type entity (INCLUDES). This verb-type entity is a *gerund*. A gerund is a verb or action word that takes a noun form. A 1:M relationship exists between each of the noun-type entities and the verb-type entity. A M:N relationship is a pair of 1:M relationships, one in either direction. The verb-type entity (INCLUDES) captures the data that results from the two noun-type entities interacting with each other.

Entities in an ERD eventually become tables in a database. You use relationships between data items and groups of data items to test the level of normalization in a database design.

Primary and Foreign Keys

A primary key (pkey) is an attribute (column) in a table that serves a special purpose. The data items that make up this attribute are unique; no two data item values are the same. The pkey value serves to uniquely identify each row in the table. You can select a row by its pkey value for an operation. Although the SQL Server environment doesn't enforce the presence of a pkey, Microsoft strongly advises that each table have an explicitly designated pkey.

Each table has only one pkey, but the pkey can include more than one attribute. You can create a pkey for the table in Screen 2 by combining CustID, OrderID, and ProductName. We call this combination a concatenated pkey.

A foreign key (fkey) is an attribute that forms an implied link between two tables that are in a 1:M relationship. The fkey, which is a column in the table of the many, is usually a pkey in the table of the one. Fkeys represent a type of controlled redundancy.

First Normal Form

You now have a basic familiarity with data, relationships, information, and business rules. Now let's understand the first three levels of normalization.

Screen 2 shows a simulated, poorly normalized table, which I'll call AllData. AllData is data from the OrderEntry database that I have reorganized for this example. A fictitious manager of a sporting goods supply company requested one order, customer identity, the order's shipping destination, billing address, contact phone number, placement time of the order, how and when delivery occurred, what articles were in the order, and which of the company's employees was the key person in the sale. Screen 2 shows the data rows in the ORDER table in a poorly normalized database. The level of redundancy is high.

When you normalize, you start from the general and work toward the specific, applying certain tests along the way. Users call this process decomposition. Decomposition eliminates insertion, update, and deletion anomalies; guarantees functional dependencies; removes transitive dependencies, and reduces non-key data redundancy. We'll decompose the sample data in Screen 2 from First Normal Form (1NF) through Second Normal Form (2NF) into Third Normal Form (3NF).

For a table to be in 1NF you need to ensure that the data is *atomic,* having no repeating groups. A concatenated pkey characterizes a 1NF table.

Atomic data is a form of minimalism for data items. A data item is *atomic* if only one item is in each cell of a table. For example, in the AllData table in Screen 2, the attribute ShippedTo encompasses a street address, a city, a state, a postal code, and a country abbreviation. To render this data atomic, you separate this single attribute into several—ShipAddr, ShipCity, ShipState, ShipZip, and ShipCountry. You must do the same separation with the BilledTo attribute. Then, the data in the AllData table is atomic.

Repeating groups are cells that have more than one occurrence. In a programming language, this concept is an array. For instance, if this database supported repeating groups (which it does not because it is a relational database), you would see a single row for this order with a repeating group for ProductName and QuantityPurchased. The set of these two columns would occur five times for this one order, once for each product purchased, thereby minimizing the redundancy in the new version of table AllData. This minimalism might work for some nonrelational database architectures and file-processing schemes, but the relational model precludes having repeating groups.

After you designate a pkey, table AllData will be in 1NF. In the description of pkeys, I suggested that a pkey for the AllData table is CustID + OrderID + ProductName. Let's designate that combination as the concatenated pkey.

Toward 2NF

2NF is a condition of *full functional dependency* on the whole pkey; the pkey must determine each non-pkey attribute. 1NF requires that a table have a pkey, and we have designated the combination of CustID + OrderID + ProductName for that role. To test for functional dependency, let's see whether the pkey determines each non-pkey attribute.

For each non-key attribute, you proceed as follows. What determines the CompanyName? One of our business rules says that each company has a Customer ID (CustID), and the CustID represents the company and each of its related attributes (CompanyName, CustomerContact, ContactPhone). However, in table AllData, does CustID + OrderID + ProductName determine CompanyName? Does CompanyName depend on what it bought and when? No. Therefore, CompanyName is not fully functionally dependent on the whole pkey.

As you test each non-key attribute against the known business rules, you can see that CustID defines some non-key attributes, OrderID defines other non-pkey attributes, and ProductName defines still other non-pkey attributes. 2NF says that all non-pkey attributes must be fully functionally dependent on the whole pkey. You must modify table AllData to make it 2NF.

If you created three tables, each of which had as its pkey a single attribute of the AllData concatenated pkey, you would have at least part of AllData in 2NF. The solution would look like Screen 3 .

The new Customer table has greatly reduced the redundant data present in table AllData. The new Order table still has a high level of redundancy, which we can correct by further decomposition. We have completely normalized the new Product table.

Determining 3NF

You achieve 3NF when you have resolved all *transitive dependencies*. Once again, you'll have to test the attributes in each table, but this time you test to see whether, within a table, any non-key attribute determines the value of another non-key attribute. Such a determination defines transitive dependency. A *transitive dependency* causes additional redundancy, which Screen 3 illustrates in the Order table.

Let's start with the Order table to analyze transitive dependencies. One business rule states that each order will have a unique order identifier. An order occurs when a customer purchases one or many products on a given day, at a given time. Therefore, attribute OrderDate is fully functionally dependent on OrderID. But what determines ShippingDate or ShippingMethod? Does OrderID determine the product and the shipping destination? The business rules will have to answer all these questions. For instance, OrderDate might affect ShippingDate. Having the ordered product (ProductName) in stock might also affect ShippingDate. A combination of OrderID and ProductName affect QuantityPurchased. OrderID and CustID affect the shipping address attributes (ShipAddr, ShipCity, ShipState, ShipZip, and ShipCountry).

The Customer table includes some transitive dependencies. The table recognizes a business rule that determines whether the customer is the company (CompanyName attribute). But, does CustID determine the CustomerContact? What if this company has more than one CustomerContact on file? If so, do you need to repeat all the billing address data for the second and third contact? Your company can institute a rule that allows only one contact person per customer, but from a salesperson's perspective, this rule would be restrictive. The salesperson and the retailer want to sell product and services. Why would they want a rule that would hamper this goal?

Screen 4 is a 3NF version of AllData, because each of the tables in Screen 4 meets the criteria for 3NF:

- Each table is a flat file, or spreadsheet format, with all-atomic data items, no repeating groups, and a designated pkey.
- Each table has all non-pkey attributes fully functionally dependent on the whole pkey.
- All transitive dependencies are removed from each table.

You still have to cross-reference the data from one table to the data in another table. Using cross-referencing, adding the second order to the

Order table will let you know what that order included (OrderDetail table).

Normalizing the Database

Now that you have decomposed the AllData table into seven smaller tables, you need to cross-reference the seven tables. You have reduced the level of data redundancy and can now fit more rows of any one table on a single block for physical reads and writes. However, what good is this organization if you have no way of relating one table to another?

In the process of reorganizing the data into the new set of tables, you reviewed the business rules. The business rules define data interrelationships:

- A CUSTOMER has many associated CUSTOMER_CONTACTs, but a CUSTOMER_CONTACT works for only one CUSTOMER at a time (1:M, CUSTOMER:CUSTOMER_CONTACT).
- A CUSTOMER can have as many ADDRESSes on file as necessary; each ADDRESS relates to one and only one CUSTOMER (1:M, CUSTOMER: ADDRESS).
- A CUSTOMER can place many ORDERs; each ORDER points back to one and only one CUSTOMER (1:M, CUSTOMER:ORDER).
- A SALESPERSON is responsible for many ORDERs; each ORDER is credited to one SALESPERSON (1:M, SALESPERSON:ORDER).
- Each ORDER can contain one or many ORDER_DETAILs (items ordered); each ORDER_ DETAIL relates back to one ORDER (1:M, ORDER: ORDER_DETAIL).
- A PRODUCT can be a participant in many ORDER_DETAILs; each ORDER_DETAIL points back to one and only one PRODUCT (1:M, PRODUCT:ORDER_DETAIL).

For each 1:M relationship, you take the pkey of the one and embed it as an fkey in the table of the many. Screen 5 shows the result. In some of these tables, I've concatenated the fkey to the existing pkey to enhance the design flexibility and functionality.

You can analyze each table independently of all others in the database, and then deduce a normal form for that table. However, the success of the database design and normalization hinges on what kind of relationship each table has to each other table, and on the correct expression of each relationship.

If you ensure that each table is in 3NF, you avoid problems that can arise when users update data. However, look at all data attributes across the database, and evaluate the normal form of the entire database. You must make sure that you've stored each non-pkey attribute only once to remove all redundant data and to remove the possibility of unsynchronized data, which can damage data recovery.

Accuracy and Performance

A poorly normalized database and poorly normalized tables can cause problems ranging from excessive disk I/O and subsequent poor system performance to inaccurate data. An improperly normalized condition can result in extensive data redundancy, which puts a burden on all programs that modify the data.

From a business perspective, the expense of bad normalization is poorly operating systems and inaccurate, incorrect, or missing data. Applying normalization techniques to OLTP database design helps create efficient systems that produce accurate data and reliable information.

Related: Pivoting Data

Source URL: https://www.itprotoday.com/microsoft-sql-server/sql-design-why-you-need-database-normalization