The great primary-key debate

by | More from | *Published: 3/22/02*

By definition, a relational database must contain normalized tables, and to be properly normalized, a table must contain a primary key. Database developers often disagree about whether it's better to use naturally occurring data or meaningless values as a table's primary key. Technically, there is no right or wrong to this debate—only very strong opinions. We strongly support the creation of primary keys from meaningless values, which we'll refer to as surrogate keys. In this article, we discuss the strengths of surrogate keys and the inherent weaknesses of natural keys.

Primary key defined

Before we enter the debate, we define the term primary key. A primary key uniquely identifies each record within a table, but that's only half the story. The main purpose of a primary key is to relate records to additional data stored in other tables. In this sense, the primary key is a simple pointer between related records in different tables. As such, the primary-key value has no meaning to the user and no association to the data to which it's assigned.

The developer must apply a few rules when choosing a primary key for each table:

- The primary key must uniquely identify each record.

- A record's primary-key value can't be null.
- The primary key-value must exist when the record is created.
- The primary key must remain stable—you can't change the primary-key field(s).
- The primary key must be compact and contain the fewest possible attributes.
- The primary-key value can't be changed.

Note that the word must in the above list doesn't mean perhaps or most of the time—must is absolute. That doesn't mean you can't break the rule; it just means your application won't adhere to the relational database model if you choose to break these established rules. In addition, even though relational database theory requires a particular condition, don't expect the relational database system to enforce it as you're developing your application—that's up to you. For more complete information on the relational database model, normalization, and primary keys, visit the following Web sites:

- Database Normalization – Definitions
- The University of Texas at Austin

Natural keys break the rules

Now, let's see how natural keys stand up to the above rules. To fulfill relational database rules, a table can't contain duplicate records. That means at least one attribute must uniquely identify each record. Let's suppose a table stores employee information, including the employee's name, birth date, Social Security number, and hire date. There should be only one record for each employee.

Initially, the name appears to be the perfect candidate for a natural primary key, but let's take a closer look. For starters, you can't create the record until you know the employee's name, because the primary key can't be null. Therefore, you can't enter any of the employee's data until the proper name is known. (You would probably avoid entering the data without the name anyway, but that's not always the case. For instance, Social Security number might be another good candidate for a natural primary key, but that piece of information is often the last to be tracked down.)

Primary keys can consist of more than one field, so storing the first and last name separately isn't a problem. However, the name might not be unique to your company, and even if it is, it might not remain that way. As soon as you have two employees with the same name, the primary key must be expanded to at least three fields, and you find yourself breaking a rule.

Next

Jump to Page: 1 2

The great primary-key debate

Correcting errors

Let's further complicate the example by supposing that you enter the name incorrectly. Initially, this doesn't seem like such a big problem; you simply correct the value. But remember, you're not supposed to change the primary-key value. Doing so often violates referential integrity if there are related records. With referential integrity features enabled, you can usually update a primary-key value, and the data engine will update related values automatically. But just because you can update a primary-key value doesn't mean you should, and you definitely shouldn't allow the uneducated user to do so. A primary-key value shouldn't be subject to data entry errors, because changing the value violates a rule.

In our example, correcting a misspelled name probably won't have too many repercussions. But suppose your primary key is based on a purchase-order number. When you try to correct an incorrectly entered value, you might learn that the correct purchase-order number already exists. Now you've got a problem that your application's referential integrity feature isn't equipped to resolve. Any time a primary key depends on external data, you risk typographical errors.

Other problems can crop up with a key based on a purchase order or similar information. A change in business could change the value's format by adding or deleting characters or even completely changing the nature of the value. This type of change is dictated by business and is beyond your control. Imagine your boss' surprise when he or she learns your order and inventory application can't integrate a new company policy on order numbers without a complete overhaul. For this reason, we recommend you select a value that you can control and maintain. You will never be able to control natural data.

Surrogate keys comply with the rules

Clearly, using a natural key as your primary key poses more than a few problems. Now, consider an incrementing value field as the employee table's primary key and see how it measures up. (You can use an expression to create the incrementing value or depend on the database program if it has an auto-incrementing data type.) Most importantly, the value will always be unique. Since the system generates the value, you'll avoid data entry (and other human) errors. In addition, the value will always exist at the time the record is entered, so the primary-key value will never be null.

A surrogate key is immune to changes in business. In addition, the key depends on only one field, so it's compact. The auto-incrementing field provides a unique, stable, and compact primary key.

Other arguments debunked

Many database development systems apply a unique index to a primary key, which eliminates duplicate records—the system simply won't accept a duplicate. However, you can apply a unique index manually; you don't need a primary key for indexing. A manually applied index will consume a little more overhead than the primary key's index, but it is well worth the small drain on your resources.

Some developers think a primary key should identify the record by association. In other words, the user should readily recognize that the primary-key value "Jane Smith" relates to the record for the employee Jane Smith. If the primary-key value for Jane Smith is a meaningless value, such as an auto-incremented value, there's no way to associate that value to Jane Smith's information. The truth is, no rule requires any association between the primary-key value and the record.

In a well-designed database, users never need to see a primary-key value. In fact, a user need never know the primary key even exists. Used correctly (to establish relationships), primary-key values are useless to the user, since your application maintains the relationships behind the scenes. In fact, surrogate keys work well precisely because there isno association between the value and the record. No matter what happens to the business or the entity, the surrogate key remains neutral.

Keeping score

When comparing the two types of keys side by side, natural keys lose, as **Table A** *demonstrates. Data is just that—data. Data shouldn't be used as a system pointer, because these items are subject to input error and are beyond the control of the developer. Programmatically or system-generated keys are stable, they're not subject to input errors, and they're never null. They provide the perfect pointer to related data.*

Table A

Rule	Natural Key	Surrogate Key
The primary key must uniquely identify each record.	••••• *But subject to input and other human errors*	••••• *System-generated value is always unique*
The primary key can't be null.	••• *Can't enter record until*	••••• *Generated by system*

	value is known	*when record is created*
The primary key must exist when the record is created.	••• *Can't enter record until value is known*	••••• *Generated by system when record is created*
The primary key must remain stable—you can't change the primary-key field(s).	• *Natural keys are subject to business rules and other outside influences.*	••••• *Surrogate keys are neutral to the application's function and the data.*
The primary key must be compact and contain the fewest possible attributes.	••• *A natural key can consist of many fields.*	••••• *Surrogate key is always just one field*
The primary-key value can't be changed.	• *Natural data often changes.*	••••• *No reason to change a meaningless value*
• *Doesn't comply uu* ••• *Often fails, but can be done uu* ••••• *Always meets requirement*		

Natural vs. surrogate keys

The single-most important issue facing the database developer is good design. If the foundation is weak, so is the building. To avoid future problems and subsequent (and perhaps convoluted) repairs, we recommend that you use surrogate keys. This simple design choice is one of the easiest ways to provide your application with a strong, stable, yet flexible foundation.

Prev

Jump to Page: 1 2