Agile Data

Choosing a Primary Key: Natural or Surrogate?

@scottwambler

Home Roles Practices Road maps Resources #AgileDB Contact us

Search

This article overviews strategies for assigning primary keys to a table within a relational database. In particular, it focuses on the issue of when to use natural keys and when to use surrogate keys. Some people will tell you that you should always use natural keys and others will tell you that you should always use surrogate keys. These people invariably prove to be wrong, typically they're doing little more than sharing the prejudices of their "data religion" with you. The reality is that natural and surrogate keys each have their advantages and disadvantages, and that no strategy is perfect for all situations. In other words, you need to know what you're doing if you want to get it right. This article discusses:

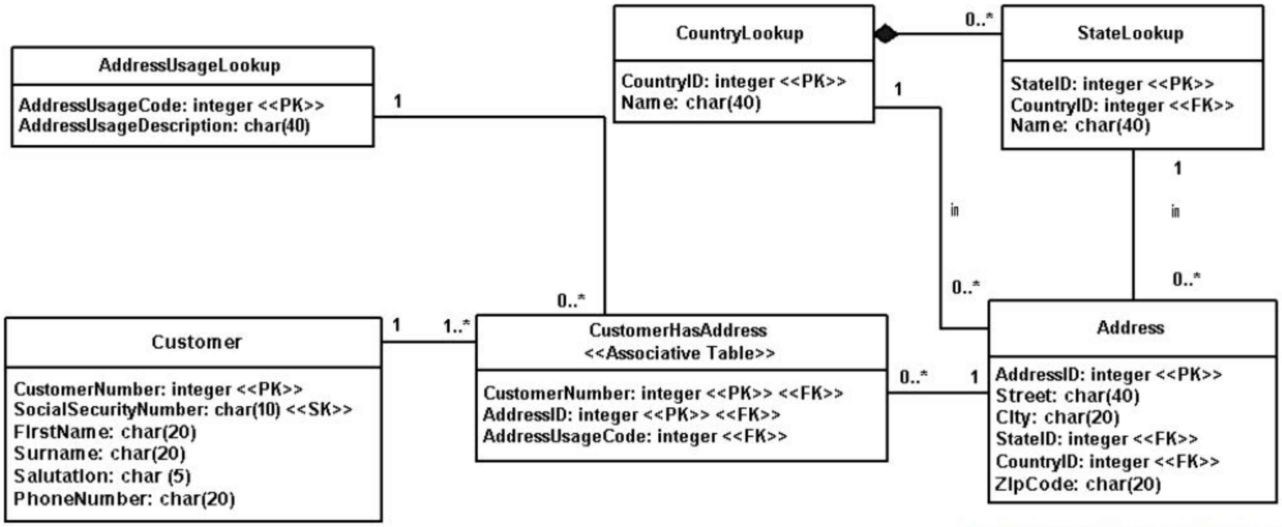
1. Common Key Terminology

Let's start by describing some common terminology pertaining to keys and then work through an example. These terms are:

- **Key**. A key is one or more data attributes that uniquely identify an entity. In a physical database a key would be formed of one or more table columns whose value(s) uniquely identifies a row within a relational table.
- **Composite key**. A key that is composed of two or more attributes.
- **Natural key**. A key that is formed of attributes that already exist in the real world. For example, U.S. citizens are issued a Social Security Number (SSN) that is unique to them (this isn't guaranteed to be true, but it's pretty darn close in practice). SSN could be used as a natural key, assuming privacy laws allow it, for a *Person* entity (assuming the scope of your organization is limited to the U.S.).
- **Surrogate key**. A key with no business meaning.
- **Candidate key**. An entity type in a logical data model will have zero or more candidate keys, also referred to simply as unique identifiers (note: some people don't believe in identifying candidate keys in LDMs, so there's no hard and fast rules). For example, if we only interact with American citizens then SSN is one candidate key for the *Person* entity type and the combination of name and phone number (assuming the combination is unique) is potentially a second candidate key. Both of these keys are called candidate keys because they are candidates to be chosen as the primary key, an alternate key or perhaps not even a key at all within a physical data model.
- **Primary key**. The preferred key for an entity type.
- **Alternate key**. Also known as a secondary key, is another unique identifier of a row within a table.
- **Foreign key**. One or more attributes in an entity type that represents a key, either primary or secondary, in another entity type.

Figure 1 presents a physical data model (PDM) for a physical address using the UML notation. In Figure 1 the *Customer* table has the *CustomerNumber* column as its primary key and *SocialSecurityNumber* as an alternate key. This indicates that the preferred way to access customer information is through the value of a person's customer number although your software can get at the same information if it has the person's social security number. The *CustomerHasAddress* table has a composite primary key, the combination of *CustomerNumber* and *AddressID*. A foreign key is one or more attributes in an entity type that represents a key, either primary or secondary, in another entity type. Foreign keys are used to maintain relationships between rows. For example, the relationships between rows in the *CustomerHasAddress* table and the *Customer* table is maintained by the *CustomerNumber* column within the *CustomerHasAddress* table. The interesting thing about the *CustomerNumber* column is the fact that it is part of the primary key for *CustomerHasAddress* as well as the foreign key to the *Customer* table. Similarly, the *AddressID* column is part of the primary key of *CustomerHasAddress* as well as a foreign key to the *Address* table to maintain the relationship with rows of *Address*.

Figure 1. A simple PDM modeling Customer and Address.


CountryLookup
CountryID: integer <<PK>>
Name: char(40)
StateLookup
StateID: integer <<PK>>
CountryID: integer <<FK>>
Name: char(40)
AddressUsageLookup
AddressUsageCode: integer <<PK>>
AddressUsageDescription: char(40)
Customer
CustomerNumber: integer <<PK>>
SocialSecurityNumber: char(10) <<SK>>
FirstName: char(20)
Surname: char(20)
Salutation: char (5)
PhoneNumber: char(20)
CustomerHasAddress
<<Associative Table>>
CustomerNumber: integer <<PK>> <<FK>>
AddressID: integer <<PK>> <<FK>>
AddressUsageCode: integer <<FK>>
Address
AddressID: integer <<PK>>
Street: char(40)
City: char(20)
StateID: integer <<FK>>
CountryID: integer <<FK>>
ZipCode: char(20)
0..*
1
1
in
in
0..*
0..*
1
1
1..*
0..*
1
0..*

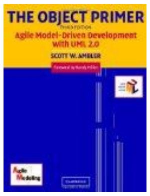
2. Comparing Natural and Surrogate Key Strategies

There are two strategies for assigning keys to tables:

1. **Natural keys**. A natural key is one or more existing data attributes that are unique to the business concept. For the *Customer* table there was two candidate keys, in this case *CustomerNumber* and *SocialSecurityNumber*.

2. **Surrogate key**. Introduce a new column, called a surrogate key, which is a key that has no business meaning. An example of which is the *AddressID* column of the *Address* table in Figure 1. Addresses don't have an "easy" natural key because you would need to use all of the columns of the *Address* table to form a key for itself (you might be able to get away with just the combination of *Street* and *ZipCode* depending on your problem domain), therefore introducing a surrogate key is a much better option in this case.

The advantage of natural keys is that they exist already, you don't need to introduce a new "unnatural" value to your data schema. However, the disadvantage of natural keys is that because they have business meaning they are effectively coupled to your business: you may need to rework your key when your business requirements change. For example, if your users decide to make *CustomerNumber* alphanumeric instead of numeric then in addition to updating the schema for the *Customer* table (which is unavoidable) you would have to change every single table where *CustomerNumber* is used as a foreign key.



There are several advantages to surrogate keys. First, they aren't coupled to your business and therefore will be easier to maintain (assuming you pick a good implementation strategy). For example, if the *Customer* table instead used a surrogate key then the change would have been localized to just the *Customer* table itself (*CustomerNumber* in this case would just be a non-key column of the table). Of course, if you needed to make a similar change to your surrogate key strategy, perhaps adding a couple of extra digits to your key values because you've run out of values, then you would have the exact same problem. Second, a common key strategy across most, or better yet all, tables can reduce the amount of source code that you need to write, reducing the total cost of ownership (TCO) of the systems that you build. The fundamental disadvantage of surrogate keys is that they're often not "human readable", making them difficult for end users to work with. The implication is that you might still need to implement alternate keys for searching, editing, and so on.

The fundamental issue is that keys are a significant source of coupling within a relational schema, and as a result they are difficult to change. The implication is that you generally want to avoid keys with business meaning because business meaning changes. Having said that, I have a tendency to use natural keys for lookup/reference tables, particularly when I suspect that the key values won't change any time soon, as I describe below. Fundamentally, there is no clear answer as to whether or not you should prefer natural keys over surrogate keys, regardless of what the zealots on either side of this religious argument may claim, and that your best strategy is be prepared to apply one strategy or the other whenever it makes sense.

3. Surrogate Key Implementation Strategies

There are several common options for implementing surrogate keys:

1. **Key values assigned by the database**. Most of the leading database vendors – companies such as Oracle, Sybase, and Informix – implement a surrogate key strategy called incremental keys. The basic idea is that they maintain a counter within the database server, writing the current value to a hidden system table to maintain consistency, which they use to assign a value to newly created table rows. Every time a row is created the counter is incremented and that value is assigned as the key value for that row. The implementation strategies vary from vendor to vendor, sometimes the values assigned are unique across all tables whereas sometimes values are unique only within a single table, but the general concept is the same.
2. **MAX() + 1**. A common strategy is to use an integer column, start the value for the first record at 1, then for a new row set the value to the maximum value in this column plus one using the SQL MAX function. Although this approach is simple it suffers from performance problems with large tables and only guarantees a unique key value within the table.
3. **Universally unique identifiers (UUIDs)**. UUIDs are 128-bit values that are created from a hash of the ID of your Ethernet card, or an equivalent software representation, and the current datetime of your computer system. The algorithm for doing this is defined by the Open Software Foundation (www.opengroup.org).
4. **Globally unique identifiers (GUIDs)**. GUIDs are a Microsoft standard that extend UUIDs, following the same strategy if an Ethernet card exists and if not then they hash a software ID and the current datetime to produce a value that is guaranteed unique to the machine that creates it.
5. **High-low strategy**. The basic idea is that your key value, often called a persistent object identifier (POID) or simply an object identified (OID), is in two logical parts: A unique HIGH value that you obtain from a defined source and an N-digit LOW value that your application assigns itself. Each time that a HIGH value is obtained the LOW value will be set to zero. For example, if the application that you're running requests a value for HIGH it will be assigned the value 1701. Assuming that N, the number of digits for LOW, is four then all persistent object identifiers that the application assigns to objects will be combination of 17010000,17010001, 17010002, and so on until 17019999. At this point a new value for HIGH is obtained, LOW is reset to zero, and you continue again. If another application requests a value for HIGH immediately after you it will given the value of 1702, and the OIDs that will be assigned to objects that it creates will be 17020000, 17020001, and so on. As you can see, as long as HIGH is unique then all POID values will be unique.

The fundamental issue is that keys are a significant source of coupling within a relational schema, and as a result they prove difficult to refactor. The implication is that you want to avoid keys with business meaning because business meaning changes. However, at the same time you need to remember that some data is commonly accessed by unique identifiers, for example customer via their customer number and American employees via their Social Security Number (SSN). In these cases you may want to use the natural key instead of a surrogate key such as a UUID or POID.

4. Tips for Effective Keys

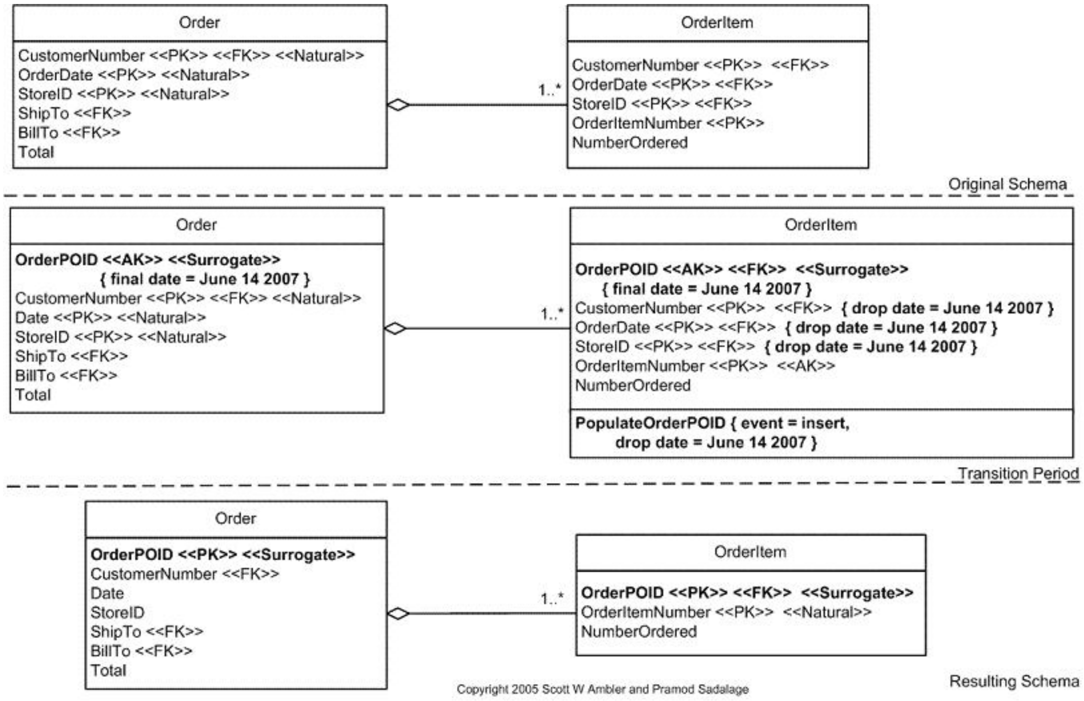
How can you be effective at assigning keys? Consider the following tips:

1. **Avoid "smart" keys**. A "smart" key is one that contains one or more subparts which provide meaning. For example the first two digits of an U.S. zip code indicate the state that the zip code is in. The first problem with smart keys is that have business meaning. The second problem is that their use often becomes convoluted over time. For example some large states have several codes, California has zip codes beginning with 90 and 91, making queries based on state codes more complex. Third, they often increase the chance that the strategy will need to be expanded. Considering that zip codes are nine digits in length (the following four digits are used at the discretion of owners of buildings uniquely identified by zip codes) it's far less likely that you'd run out of nine-digit numbers before running out of two digit codes assigned to individual states.
2. **Consider assigning natural keys for simple "look up" tables**. A "look up" table is one that is used to relate codes to detailed information. For example, you might have a look up table listing color codes to the names of colors. For example the code 127 represents "Tulip Yellow". Simple look up tables typically consist of a code column and a description/name column whereas complex look up tables consist of a code column and several informational columns.
3. **Natural keys don't always work for "look up" tables**. Another example of a look up table is one that contains a row for each state, province, or territory in North America. For example there would be a row for California, a US state, and for Ontario, a Canadian province. The primary goal of this table is to provide an official list of these geographical entities, a list that is reasonably static over time (the last change to it would have been in the late 1990s when the Northwest Territories, a territory of Canada, was split into Nunavut and Northwest Territories). A valid natural key for this table would be the state code, a unique two character code – e.g. CA for California and ON for Ontario. Unfortunately this approach doesn't work because Canadian government decided to keep the same state code, NW, for the two territories.
4. **Your applications must still support "natural key searches"**. If you choose to take a surrogate key approach to your database design you mustn't forget that your applications must still support searches on the domain columns that still uniquely identify rows. For example, your *Customer* table may have a *Customer_POID* column used as a surrogate key as well as a *Customer_Number* column and a *Social_Security_Number* column. You would likely need to support searches based on both the customer number and the social security number. Searching is discussed in detail in Best Practices for Retrieving Objects from a Relational Database.
5. **Don't naturalize surrogate keys**. As soon as you display the value of a surrogate key to your end users, or worse yet allow them to work with the value (perhaps to search), you have effectively given the key business meaning. This in effect naturalizes the key and thereby negates some of the advantages of surrogate keys.

5. What to Do When You Make the "Wrong" Choice

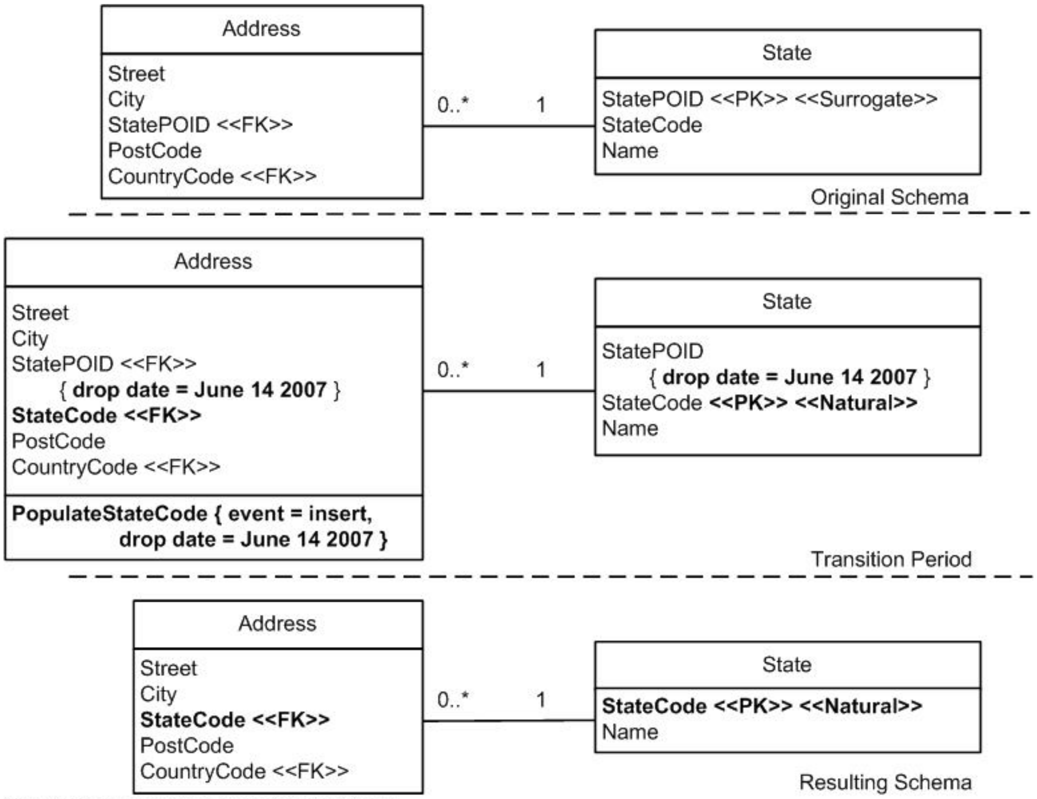
First of all, don't worry about this: You're only human, and no matter how good you are at database design you're going to make mistakes. The good news is that as I show in The Process of Database Refactoring it is possible, albeit it may require a bit of work, to replace a natural key with a surrogate key (or vice versa). To replace a natural key with a surrogate you would apply the Introduce Surrogate Key refactoring, as you see depicted in Figure 2 to replace the key of the Order table. To replace a surrogate key with a natural key you would apply the Replace Surrogate Key with Natural Key refactoring, as you see in Figure 3 to replace the key of the State table.

Figure 2. Replacing the natural key of the Order table.


Order
CustomerNumber <<PK>> <<FK>> <<Natural>>
OrderDate <<PK>> <<Natural>>
StoreID <<PK>> <<Natural>>
ShipTo <<FK>>
BillTo <<FK>>
Total
1..*
OrderItem
CustomerNumber <<PK>> <<FK>>
OrderDate <<PK>> <<FK>>
StoreID <<PK>> <<FK>>
OrderItemNumber <<PK>>
NumberOrdered
Original Schema
Order
OrderPOID <<AK>> <<Surrogate>>
{ final date = June 14 2007 }
CustomerNumber <<PK>> <<FK>> <<Natural>>
Date <<PK>> <<Natural>>
StoreID <<PK>> <<Natural>>
ShipTo <<FK>>
BillTo <<FK>>
Total
1..*
OrderItem
OrderPOID <<AK>> <<FK>> <<Surrogate>>
{ final date = June 14 2007 }
CustomerNumber <<PK>> <<FK>> { drop date = June 14 2007 }
OrderDate <<PK>> <<FK>> { drop date = June 14 2007 }
StoreID <<PK>> <<FK>> { drop date = June 14 2007 }
OrderItemNumber <<PK>> <<AK>>
NumberOrdered
PopulateOrderPOID { event = insert,
drop date = June 14 2007 }
Transition Period
Order
OrderPOID <<PK>> <<Surrogate>>
CustomerNumber <<FK>>
Date
StoreID
ShipTo <<FK>>
BillTo <<FK>>
Total
1..*
OrderItem
OrderPOID <<PK>> <<FK>> <<Surrogate>>
OrderItemNumber <<PK>> <<Natural>>
NumberOrdered
Resulting Schema

Copyright 2005 Scott W Ambler and Pramod Sadalage

Figure 3. Replacing the surrogate key within the State table.



Copyright 2005 Scott W Ambler and Pramod Sadalage

Let Us Help