Natural versus Surrogate Keys. Performance and Usability

Dragos-Paul POP
Academy of Economic Studies, Bucharest, ROMANIA
dragos_paul_pop@yahoo.com

Choosing the primary key for a table proves to be one of the most important steps in database design. But what happens when we have to pick between a natural or a surrogate key? Is there any performance issue that we must have in mind? Does the literature have a preferred pick? Is usability a concern? We'll have a look at the advantages and disadvantages of both natural and surrogate keys and the performance and usability issues they address.
Keywords: *primary keys, natural keys, surrogate keys, superkey, candidate key, unique key, performance, usability.*

1 Introduction

Choosing a primary key is really important because it affects the database at the performance and usability levels. The literature speaks of both natural and surrogate keys and gives reasons for choosing one over the other.

Before we get to talk about natural and surrogate keys in a relational transactional table, we must define a few other key concepts used in the relational database model architecture. The concepts to be defined are the superkey, the candidate key, the unique key and the primary key.

2 Key concepts

The superkey is defined as being a set of attributes of a given table that verifies one main condition. The condition in question is that there are no two distinct tuples in that table with an identical value for the superkey set. Also, the attributes comprising the superkey set are said to be functionally dependent. This makes true the following statement: if S is a superkey for the relation R, than the cardinality of the projection of R over S is the same as the cardinality of R. After a table has gone through normalization, we can say that all its attributes form a superkey, because there are no two tuples that are identical for all the values of the set.

From the superkey concept, we can define the candidate key. This is sometimes called a minimal superkey, because a candidate key is, in fact, a minimal set of attributes necessary to uniquely identify a tuple. In other words, a set of attributes is said to be a candidate key if there are no two tuples with the same value for the key and there is no other subset of these attributes that can form a candidate key. This is where the "minimal" property of the candidate key derives from. A table can have multiple candidate keys.

Another concept related to the superkey is the unique key. Again, just like a superkey and a candidate key, the unique key can uniquely identify each row in a table. Although this is not a rule, unique keys tend to only comprise a single column. The difference between candidate keys and unique keys is that, in practice, unique keys do not enforce the NOT NULL constraint. This means they can contain the NULL value and still uniquely identify table rows. Why? Because of the way NULL is treated by the database management systems. NULL is not a value, but the absence of a value, so the unique key concept holds true even for rows with NULL for the unique key. This is because identification of two equal keys is done based on their values, and since NULL is not a value, two keys containing NULL are not considered to be equal. This is by no means to be taken as a rule, because it differs in implementation across database management systems. A better definition of a unique key is that two

tuples cannot have the same value for the unique key if NULL values are not used. So, a unique key only uniquely identifies rows that contain a value other than NULL for the key. As for the candidate key, a table can have multiple unique keys.

The primary key is probably the most important concept in database design. A primary key is, basically, one of the candidate keys in a table. It is a unique key that does not contain (and never will) NULL values can also be made a primary key. For some tables, even a superkey can be a primary key (but that is a little odd). So how and why is a primary key different form all the others? A table can only have one primary key and this key is the preferred way of identifying individual tuples.

3 Natural and surrogate keys

Choosing the primary key has proven to be the difficult part in database design. This is because there are two types of primary keys: natural and surrogate.

The natural key, also called a domain key or an intelligent key, is a candidate key that is logically related to the table. That is, it has business meaning, or business value. It is something that can be found in nature, it makes sense.

A natural key is a single column or set of columns that uniquely identifies a single record in a table, where the key columns are made up of real data. When I say "real data" I mean data that has meaning and occurs naturally in the world of data.

A natural key is a column value that has a relationship with the rest of the column values in a given data record. Here are some examples of natural keys values: Social Security Number, ISBN, and TaxId.

On the other hand, the surrogate key is not derived from real data; it does not have any business meaning or logic. It is a key most often generated by the database or made up using an algorithm.

A surrogate key like a natural key is a column that uniquely identifies a single record in a table. But this is where the similarity stops. They are keys that don't have a natural relationship with the rest of the columns in a table. The surrogate key is just a value that is generated and then stored with the rest of the columns in a record. The key value is typically generated at run time right before the record is inserted into a table. It is sometimes also referred to as a dumb key, because there is no meaning associated with the value. Surrogate keys are commonly a numeric number.

An important distinction between a surrogate and a primary key depends on whether the database is a current database or a temporal database. Since a current database stores only currently valid data, there is a one-to-one correspondence between a surrogate in the modeled world and the primary key of some object in the database. In this case the surrogate may be used as a primary key, resulting in the term surrogate key. In a temporal database, however, there is a many-to-one relationship between primary keys and the surrogate. Since there may be several objects in the database corresponding to a single surrogate, we cannot use the surrogate as a primary key; another attribute is required, in addition to the surrogate, to uniquely identify each object.

Authors have argued that a surrogate should have the following characteristics:

- the value is unique system-wide, hence never reused
- the value is system generated
- the value is not modifiable by the user or application
- the value contains no semantic meaning
- the value is not visible to the user or application
- the value is not composed of several values from different domains

In practice, the surrogate key is frequently a number generated by the database management system. For example, Oracle uses sequences to accomplish this task, while SQL server gives the "identity

column" option. PostgreSQL users have the "serial" option, and MySQL ones use an auto_increment attribute. Having the key independent of all other columns insulates the database relationships from changes in data values or database design (making the database more agile) and guarantees uniqueness.

4 Surrogate Key Implementation Strategies

There are several common options for implementing surrogate keys:

- Key values assigned by the database. Most of the leading database vendors – companies such as Oracle, Microsoft and IBM – 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.
- 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.
- 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).
- 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.
- 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 be 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. [3]

5 Tips for Effective Keys

How can you be effective at assigning keys? Consider the following tips, by Scott W. Ambler [3]:

- 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.
- 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.
- 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.
- 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.

- 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. [3]

6 Advantages and disadvantages

Of course, there are a lot of advantages and disadvantages of using natural or surrogate keys. Authors are divided between the two strategies. Below there is a listing of pros and cons of using both natural and surrogate keys, as Gregory A. Larsen list them:

6.1 Surrogate Key Pros and Cons

A definite design and programming aspect of working with databases is built on the concept that all keys will be supported by the use surrogate keys. To understand these programming aspects better, review these pros and cons of using surrogate keys. [4]

Pros:

- The primary key has no business intelligence built into it. Meaning you cannot derive any meaning, or relationship between the surrogate key and the rest of the data columns in a row.
- If your business rules change, which would require you to update your natural key this can be done easily without causing a cascade effect across all foreign key relationships. By using a surrogate key instead of a natural key the surrogate key is used in all foreign key relationships. Surrogate keys will not be updated over time.
- Surrogate keys are typically integers, which only require 4 bytes to store, so the primary key index structure will be smaller in size than their natural key counter parts. Having a small index structure means better performance for JOIN operations.
- It's easy to create a naming system for surrogate keys, so that remembering the primary key of a table can be made a lot easier. [4]

Cons:

- If foreign key tables use surrogate keys then you will be required to have a join to retrieve the real foreign key value. Whereas if the foreign key table used a natural key then the natural key would be already be included in your table and no join would be required. Of course this I only true if you only needed the natural key column returned in your query
- Surrogate keys are typically not useful when searching for data since they have no meaning.
- Surrogate keys have no knowledge level value. The most important function of the PK is as an interaction element between the real-world and the database. It is thorough the primary key that we usually query the database. The primary key is of fundamental importance if we are to "usefully" relate the concepts of the database to the real world. [4]

6.2 Natural Key Pros and Cons

Having natural keys as indexes on your tables mean you will have different programming considerations when building your applications. You will find that pros and cons for natural keys to be just the opposite as the pros and cons for surrogate keys. [4]

Pros:

- They already exist in the schema. There is no need for additional columns that would load the tables.
- Will require less joins when you only need to return the key value of a foreign key table. This is because the natural key will already be imbedded in your table.
- Easier to search because natural keys have meaning and will be stored in your table. Without the natural key in your table, a search for records based on a natural key would require a join to the foreign key table to get the natural key. [4]

Cons:

- Requires much more work to change a natural key, especially when foreign relationship have been built off the natural key.
- Your primary key index will be larger because natural keys are typically larger in size then surrogate keys.
- Since natural keys are typically larger in size then surrogate keys and are strings instead of integers joins between two tables on a natural key will take more time.
- Kind of hard to remember the name of the key for every table in the database [4]

7 Performance issue

The next scenario is built to test the performance of natural and surrogate keys. We will see when and if one is better than the other.

The test business logic is simple and it is about the commercial activity of a company that sells goods. The test entities are described as follows:

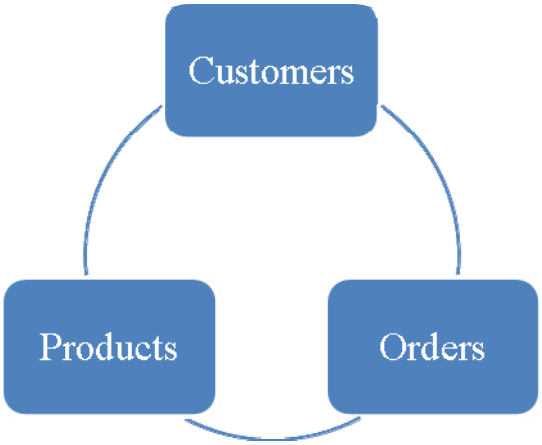

Fig 10. Database logical entities

After undergoing normalization, we get the following database structure:

- Customers
  - This table will hold all the information related to the customers, such as first and last names, email address, telephone number, address and so on
- Products
  - Here we will have details about the products that are being sold: name, price, stock etc.
- Categories
  - This table stores information about different categories of products
- Orders
  - This is the main table that holds information about customer orders, such as order date, serial number, total value
- OrderDetalis
  - The last table is used to store information about individual lines in a customer order: product, quantity, price at buying time

There are two test cases: one in which we will chose a natural key for the primary key of every table and one in which a surrogate key will be used. The tables will be loaded with data and will be tested to see the response times of simple selects and joins. The test database management systems is Oracle 10g Express Edition. The test computer is equipped with an Intel Core i5 750 processor, 4 GB of RAM and two 500

GB hard-disks at 7200 rpm connected in RAID level 0.
The test scenario uses the following table descriptions:

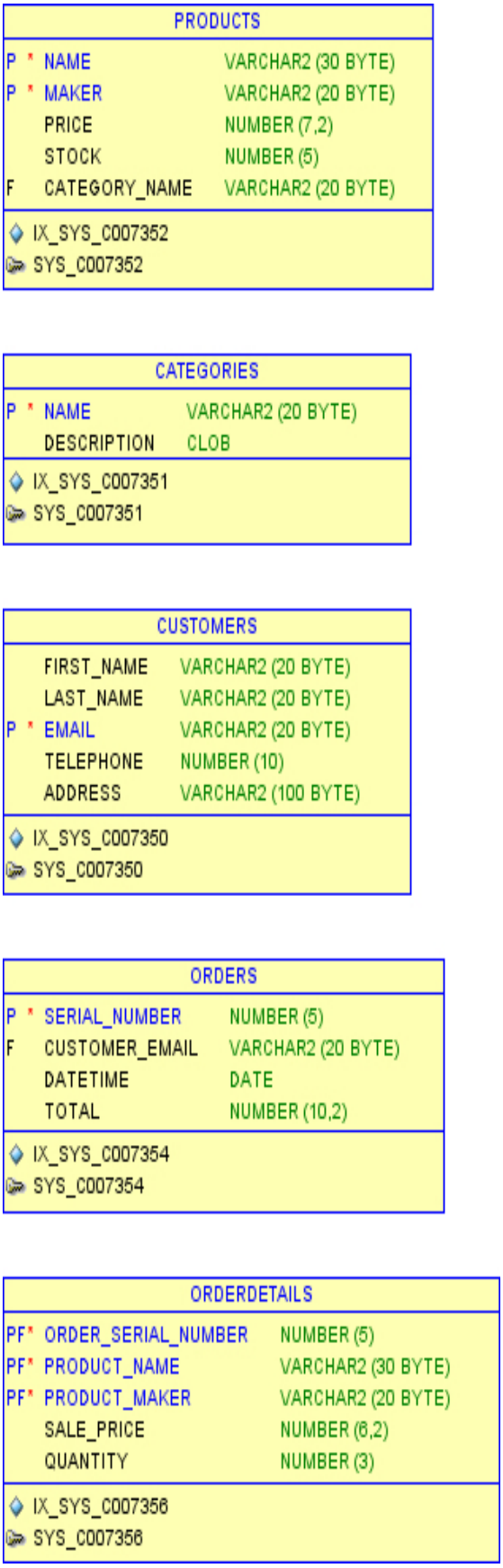

Fig. 11. Natural keys database

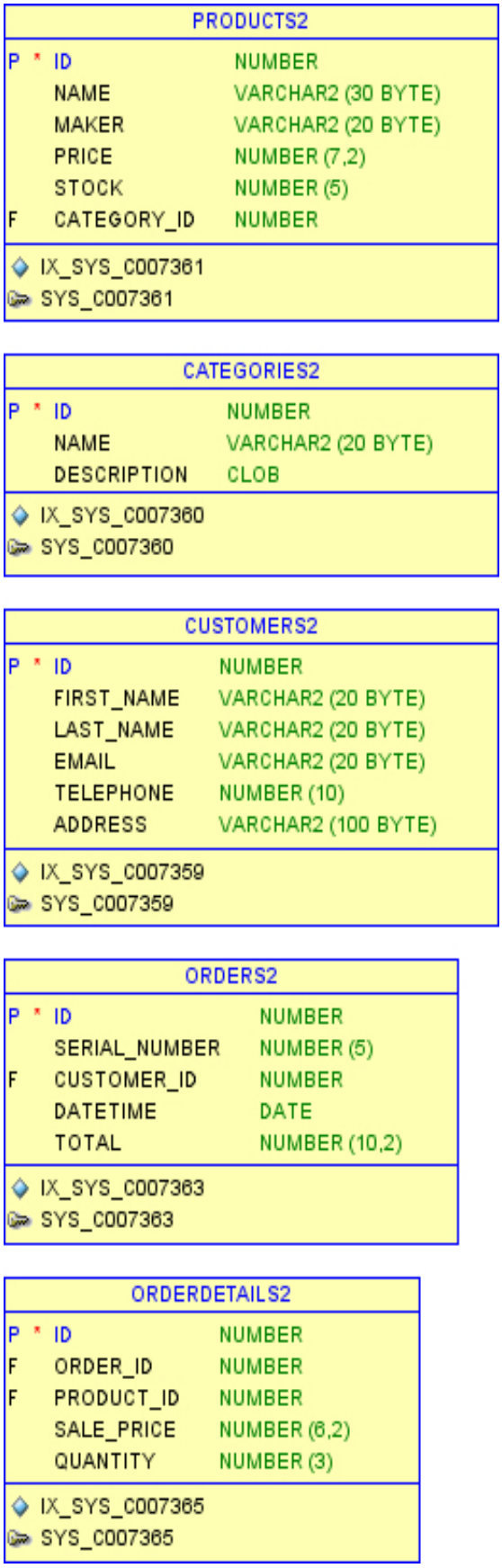

Fig. 12. Surrogate keys database

Table 1. Query results

SQL Query	Execution time
`select * from customers2, products2, categories2, orders2, orderdetails2 where customers2.id = orders2.customer_id and products2.category_id = categories2.id and orders2.id = orderdetails2.order_id and orderdetails2.product_id = products2.id`	18 ms
`select * from customers, products, categories, orders, orderdetails where customers.email = orders.customer_email and products.category_name = categories.name and orders.serial_number = orderdetails.order_serial_number and orderdetails.product_maker = products.maker and orderdetails.product_name = products.name`	20 ms
`select * from orderdetails2`	15 ms
`select * from orders2`	15 ms
`select * from categories2`	15 ms
`select * from products2`	18 ms
`select * from customers2`	14 ms
`select * from orderdetails`	17 ms
`select * from orders`	18 ms
`select * from categories`	28 ms
`select * from products`	18 ms
`select * from customers`	15 ms

8 Conclusions

As we can see from the results above, choosing surrogate keys as primary keys does not always mean adding columns to tables. Also, query times are improved, because primary indexes are smaller. This is due to the fact that surrogate keys use an integer data type, while the natural keys they replaced used a variable length character data type.

In the end, although surrogate keys tend to be better for performance, people still use natural keys just because they feel better. Generally, database designers are inclined to use surrogate keys, because making things abstract is their main issue, while application developers go with natural keys, because they have more business logic.

9

Acknowledgements

This work was cofinaced from the European Social Fund through Sectoral Operational Programme Human Resources Development 2007-2013, project number POSDRU/107/1.5/S/77213 „Ph.D. for a career in interdisciplinary economic research at the European standards".

References

[1] Breck Carter, "Intelligent Versus Surrogate Keys". Internet: http://www.bcarter.com/intsurr1.htm, October 6, 1997 [Mar 18, 2011]

[2] Michelle A. Poolet, "SQL by Design: How to Choose a Primary Key". Internet: http://www.sqlmag.com/article/systems-administrator/sql-by-design-how-to-choose-a-primary-key, April 01, 1999 [March 18, 2011]

[3] Scott W. Ambler, "Choosing a Primary Key: Natural or Surrogate?". Internet: http://www.agiledata.org/essays/keys.html, 2005 [Mar 18, 2011]

[4] Gregory A. Larsen, "SQL Server: Natural Key Verses Surrogate Key". Internet: http://www.databasejournal.com/features/mssql/article.php/3922066/SQL-Ser

[5] ver-Natural-Key-Verses-Surrogate-Key.htm, January 31, 2011 [Mar 18, 2011]

[6] Michelle A. Poolet, "Surrogate Key vs. Natural Key". Internet: http://www.sqlmag.com/article/data-modeling/surrogate-key-vs-natural-key/2, January 24, 2002 [Mar 18, 2011]

Dragos-Paul POP graduated from the Faculty of Computer Science for Business Management at the Romanian-American University of Bucharest in 2007 (Bachelor's degree) and in 2009 (Master's degree), specialising in Economic IT. He is curently a Ph.D. candidate at the Academy of Economic Studie in Bucharest. He works as an assistant teacher at the Romanian-American University in Bucahrest, teaching computer architecture, operating systems, advanced web programming and databases. His main domains of interest are web technologies, database technologies, programming languages, networking, hardware and operating system