Mike Shkolnik, Scalabium
2000

ID generation strategies

In this small article I want to describe a different methods of ID generation.
ID is an unique identifier that allows to identify a record. In relational terminology an unique identifier is called a key.

1. IDs should have no business meaning

From own experience I sure that each table must have absolutely no business meaning ID. Any column with a business meaning can potentially change and it's a fatal mistake to give your keys meaning. You can define the alternative key with your unique "business" identifier but your primary key don't must depends from some "today unique conditions" which are live today only.

Of course, to have a ZIP, telephone number, currency code or country/state abbreviation as ID is very alluring from first look, but my suggestion – don't limit own development from first step. The life will change… ZIP in each country can have the different structure, phone station can change first digits in number, currency can be changed, and country can be divided on parts…

The each primary key in one table is virtually guarantied to be used as a foreign key in other tables and when your business code will change (to add a few digits, convert the number to alphanumeric etc), you need to make a lot of changes in your database and application's sources.

Also exists the second reason of this advice. At most cases you'll join a data from few tables in your SELECT-statement. For example, you must joins the ORDER and SUPPLIER tables by ID_CUSTOMER field and selects a name and address of supplier. But the all operations with numbers (I suggest to have the ID as integers) in any database server and operating system was optimized by performance but string/date/etc operations will request a lot of additional conversion and comparisons.

Of course, the some developers can say that business ID will reduce the join using (for example, the code of country will store in SUPPLIER table as foreign key from COUNTRY table) but I must say that it's not a true because in 99.999% of common task you needs a some additional attributes from object and you'll use a joins but in this case you'll lose in performance.

2. ID uniqueness

When you assigns the ID, you must decide the two problems:
- level of uniqueness for ID
- how to obtain the ID value

Level of uniqueness is a very important but developers don't spend a needed time on solution of this task. This opinion is wrong because it's a possibility to retrieve limitations in the future life of your applications. You must select a level of uniqueness, which is needed for you in first steps of development.

There are a few levels of uniqueness that you need to consider:
- uniqueness within the table
- uniqueness within the logical partition of tables
- uniqueness across all tables

For example, the ID for customer can be unique:
- for CUSTOMER table only
- for partition of companies (CUSTOMER, SUPPLIER, CLIENT, RESTOURANT, HOTEL tables etc)
- for all tables of database

In first case the value ID=15614 can be assigned to some customer, to supplier and to invoice but all these values are different.
In second case the same value can't be assigned to some customer and supplier because these tables is from one logical partition of companies but you can assign this value to order or invoice.
In third case this value can be assigned to one record only from any table in database: either to customer or to supplier or to order or to invoice.

You must select a needed level of uniqueness in first step of development because a lot of tasks will depend from this your choice. For example, data replication or rules for semantic and/or syntactic validations.

3. Strategies for ID generation

As I wrote above the second task is how to obtain the key value. In this section I'll describe a most popular algorithms. You must select the one from them which will allows to have a great performance and efficiency in run-time of your system. For each type of system (or type of stored data) the effective method can be different and you must select it for you. You must understand that algorithms of ID generation for OLAP/warehousing and OLTP systems can be different.

PS: I suggest to select one strategy for all your tables and to generate the key values by one algorithm. Only in systems where you must have a high performance on each stage of data processing and you can't increase the performance in other steps, you must use the different strategies for other tables of same system.

PPS: also don't forget that the large system will divided on parts with few developer teams (or companies) but project leader must control the basic principles in own hands. The strategy of key generation is one from such basic principles.

3.1. **To calculate a Maximum for Integer column**

Before execution of INSERT-statement you must retrieve a MAX-value for your Integer column which was defined as Primary Key. For this task you can execute a simple SQL-statement:

```
SELECT MAX(yourPKFieldName)
FROM yourTableName
```

After that you must add one to this value and use it as the value for your key of newest record.

The problem with this algorithm is that you must lock a table before calculation of maximum value and unlock it after INSERT-statement for your newest record. In else case the some newest records can try to insert a records with same value PK (for example, in multi user systems).

Also with this algorithm you don't have unique values for IDs across all your tables (or partition), only for those stored within each table. Of course, you can execute the SQL-statements for each table and use the max-value from them but I not sure that this idea is good.

IMPORTANT:

If you'll select this method for ID generation, don't forget that you must strongly control the collisions in linked tables after deletion of some records because if you'll delete a record with ID which is maximum on some moment, the next value of MAX-value will be same as this deleted.

3.2. **Using table with key-values**

This method means that you have a some "container" in which a counter values is stored. For example, you can realize this container as some additional table.

This container can have a different structure. For example, in single row you can store a counter value for each table in assigned column, or for each table to create an additional table with one column and one row, or to use a multi-row table, where you have one row per table with two columns: identifier for table name and value for next key for this table.

The advantages of this strategy:

- you invoke a MAX-statement
- you have an unique values for all tables in one container
- for multi-row case you can lock a row/page only instead all table

The disadvantage of this strategy is that this container becomes a bottleneck. If you have a system with data, which will frequently inserted, you can cache this values in memory.

3.3. **GUIDs**

To use the GUID for ID is good idea although you can't calculate it on all platforms. From this point I don't think that GUID is a good for using in any system. Also 128bits string for Primary Key is not for me too. Of course, you can convert this string to number but don't forget that for GUID generation the operating system will read the identification number of network card in your computer and current date/time. And after that you'll run the own algorithm for conversion into number.

3.4. **In-built mechanisms of Database Servers**

The some database servers (for example, ORACLE, INFORMIX, MS SQL, INTERBASE etc) have built-in features for generation of unique values. These DBMSs uses the one from two different mechanisms:

- autoincrement type or additional flag/subtype for number type) for field definition (SERIAL type in INFORMIX or IDENTITY flag in MS SQL)
- some internal incremental sequence (SEQUENCE in ORACLE, GENERATOR in INTERBASE).

The both mechanisms allows to define a start value and step of incrementation but in first case you'll retrieve a value after successfully completed INSERT-statement (when value will be automatically generated and inserted into PK's field) and in second case you can read a next value in any time from application or some stored procedure/function without any INSERT-statement execution.

So if you want to have an unique values across partition or all tables, you can use a DBMS with sequence mechanism only.

But don't forget that first mechanism with autoincremental type allows to assign the ID without any your additional actions. In second case you must read the next value from sequence and put it into PK of new record (for example in trigger on AFTER INSERT action).

But the main disadvantage of these in-built mechanisms – they are can't be ported from one DBMS to other with small time of system rebuilding. This is not a SQL-standard: each database manufacturer have the own unique mechanism for ultimate performance and with own syntax. This can become a serious issue for you.

The main advantage – the ultimate performance and solution on database side but not on user application side. You have a guarantee that record will have a correct unique ID in any case: when user will insert a record from your application or database administrator or developer inserted this record from own tool.

3.5. **Pseudo-random values**

For ID generation you can use the pseudo-random functions. The some from them have a very high performance and uniqueness (like GUID, see 3.3), the some from them have a limited warranty of uniqueness but can satisfy you. For example, you can use the combination of user ID and current date/time with milliseconds. As alternative to user ID, you can use the workstation ID or current IP address.

This method can be useful also as additional logging system – you can always say who and when created a record – without any additional actions!

Of course, also you can use the one from mathematical algorithms for "random" value generation with high level of uniqueness.

3.6. **Strategy of segments**

When above I described the using of key-value table (see 3.2), I wrote that you must use a cache for reducing of access to container with counters. Now I want to describe the one method for caching.

The basic idea is that instead reading a key-value from container before each execution of INSERT-statement, the "reader" (some client application or session) can read the start value (for example, Integer value: 15600). And on the next few INSERT-statements (for example, 100 statements) this "reader" can generate the ID from own cache as 15601, 15602 … 15699.

When this "reader" will fill the personal segment he/she must read a next start value from container.

Of course, container must return a correct value for each segment and if you will receive the 15600 as start for next 100 records, then your "competitor reader" must receive the next segment: from 15700 to 15799.

The advantage of this strategy is that table-container is no longer as big of bottleneck and the network traffic needed to get a value for ID is very little.

Also this method can be useful for replication because you can control the segmentation per server/session.