Abstract
In this article, I will review few of database design anti-patterns and some solutions that avoid those problems. Nowadays, rare application does not have database on the backend, and while relational databases exist for a long time, it is still the most used type of database. Relational databases have often replaced other types of databases because they are easier to understand and use. On the other side, because of this easiness in the overall concept many anti-patterns are overlooked and used in many systems.

Introduction
If you know the patterns, it is very good, if you know the anti-patterns - it is even better. Anti-patterns exist everywhere, in every aspect of our lives and it is crucial to know them and use techniques and solutions to avoid them. Relational Databases exist for very long time, but still, many people use not optimized techniques to design and develop databases. It is very common for people to stick with the way they used to do it and not to question themselves: “is it the really best way to do it?” Optimization of developing techniques helps to avoid lifetime long optimization of the code.

There are many well known database anti-pattern related to database design, development and use in the application. This article will concentrate on few database design anti-patterns.

Anti-patterns
1. Assigning value to nothing (NULL)
   Null should not have any value in database logic; this approach is bad for anything, so it should not be used in database either.
2. Attribute with multi-value (list of values)
   This anti-pattern is one of the main consequences of non-normalized database design.
3. Table is not a list (super-wide tables)
   Long table is good, wide table is bad.

Anti-pattern 1:
- Name
- Assigning value to nothing (NULL)
- Problem / Bad solution
  Often databases designed with intend to use NULL for something with special meaning. While in reality, NULL means nothing then nothing cannot be something. Typical two-table database relationship is shown in Example 1.
Each category can be associated with one directory by using dir_id foreign key. If any category needs to be associated with all directories, NULL is used as the way to solve this problem. This is wrong, because this way we assigned to the NULL meaning of all directories, while NULL is nothing, it cannot have any value.

**Symptoms & Consequences**
- Using this database design will cause problems with maintenance of the database
- If NULL is already used for something, it cannot be used for its main purpose, which is do not have any value. That can lead to additional database redesign.

**Root cause**
Each attribute should have singular and unique value, not NULL. NULL cannot represent something because it is nothing. For example, the number 3 cannot represent anything else other than three. Moreover, if somebody will ever use it with different meaning, nobody will understand it. The same is for NULL in relational databases.

**Suggested solution**
There are few ways to handle this problem. Probably the easiest, but not the best, way to deal with it is to create some “special” value that will represent all directories. For example, zero or any negative number (such as -1) can be used, since usually it not used in id’s therefore it should not be in dir_id column. This will point out in more obvious way that this is some value with special meaning.

The better way is add new column to Category table that will represent relationship to all directories (true or false), while dir_id column will represent reference to only one particular directory (example 2).
Anti-pattern 2:

- **Name**
  Attribute with multi-value (list of values)

- **Problem / Bad solution**
  During design of many relational databases, the common problem is a many-to-many relationship between two tables. For example, two tables with clients and groups of clients. In the case where each client is related only to one group and each group can have many clients, no problem exists. However, if it is required to have clients related to many groups commonly comma-separated (or any other symbol separated) values used for group_id foreign key in clients table (example 3).

<table>
<thead>
<tr>
<th>Client</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>client_id</td>
<td>client_name</td>
</tr>
<tr>
<td>1</td>
<td>John Doe</td>
</tr>
<tr>
<td>2</td>
<td>Robert Smith</td>
</tr>
<tr>
<td>3</td>
<td>Mary Smith</td>
</tr>
</tbody>
</table>

Example 3

- **Symptoms & Consequences**
  - Problems with adding values to list, if we need to relate client to more groups.
  - Problem with selecting only clients related to particular group (regular expressions should be used in query)
  - Problem with getting number of clients for each group.
  - No efficient way to use indexing on that particular foreign key.

- **Root cause**
  In most cases, poor database design and lack of knowledge on good database patterns are the root cause of this antipattern. Value should be singular for each attribute.

- **Suggested solution**
  Association table should always be used for many-to-many relationships between two tables (example 4). ClientGroup table hold relationships between clients and groups.

<table>
<thead>
<tr>
<th>ClientGroup</th>
<th>Client</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>client_id</td>
<td>group_id</td>
<td>client_id</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Example 4
Anti-pattern 3:

- **Name**
  Table is not a list (super-wide tables)

- **Problem / Bad solution**
  Super-wide tables are the problem with an object that has too many attributes. Consider the table that keeps track of the purchases of insurance (example 5).

<table>
<thead>
<tr>
<th>Purchase</th>
<th>PurchasePremium</th>
</tr>
</thead>
<tbody>
<tr>
<td>purchase_id</td>
<td>client_id</td>
</tr>
<tr>
<td>5</td>
<td>1001</td>
</tr>
<tr>
<td>6</td>
<td>1002</td>
</tr>
<tr>
<td>7</td>
<td>2003</td>
</tr>
</tbody>
</table>

  Example 5

  Purchase table keeps record of different types of information about the purchase itself, such as, policy start and end dates, premium information and many other additional is possible to add here.

- **Symptoms & Consequences**
  - While it good to all info about purchase together and avoid complicated queries to retrieve data, it adds confusion to design of the database
  - Hard to maintain

- **Root cause**
  - Natural tendency to keep all information about one transaction (purchase, sale, quote, etc) together as it is in the ordered office environment
  - Belief that too many tables will complicate database design and queries

- **Suggested solution**
  To avoid this problem Purchase table in Example 4 needs to be divided into few logical tables and use the same primary key for both tables. Particular Purchase table can be divided into two or more tables. As shown in Example 6, premium data is separated from the main table.
Conclusion
In this article, just a few of many relational database anti-patterns were reviewed. Knowing the anti-patterns is very important for development, maintenance and design of databases. Good logical database schema design can save a lot of time and money on maintenance and avoid problems in future development.

References
• http://c2.com/cgi/wiki?SqlAntiPatterns
• http://en.wikipedia.org/wiki/Anti-pattern