5 Hot Tips to Avoid Creating a 'Database from Hell'

Shôn Ellerton, Jul 20, 2017

If you are about to build a new database, here are five hot tips which could help you to control your database rather than your database controlling you as it grows.

Introduction

Believe it or not, I've had a fascination with databases since I was 15, starting out with Borland's Paradox, moving over to Microsoft Access and then on to Microsoft SQL Server. For many, there is nothing particularly glossy or alluring about databases and the science of structuring data; however, for some, the need to organise data (or anything, for that matter) can be satisfying. For example, during my childhood, much to my family's bemusement, I would catalog my mountaineering hikes, stamp collection and record collection. They didn't know why, and, come for that matter, I'm not sure I understood either! However, there's probably a strong correlation between those who like organising and those who like building centralised 'single-source of truth' data repositories and abhor the practice of storing data on Excel spreadsheets.

My business has been within the mobile telecommunications and wireless sector since 1997, initially as a structural engineer and project manager and then progressively integrating my I.T. skills to developing project-tracking databases from scratch. Many of these databases grew exponentially to the point where scalability, maintenance and understandability became problematic.

In my experience, most databases start out from a simple Excel spreadsheet, which quickly becomes unusable due to limited user access, data corruption and multiple copies in existence.

In smaller groups, a likely outcome would be that a local database (e.g. Microsoft Access) be created to replace the spreadsheets; however, Microsoft Access is file-based and can be copied easily as with the spreadsheets.

In larger environments, it is often likely that either Microsoft SQL Server or Oracle be used as the database of choice, although there are plenty of others in the marketplace to choose from.

At the embryonic stage of any database, it is often the one-and-only chance to get a few fundamentals in place before the database has a chance to grow and mutate into, what I refer as a ‘Frankenstein’ or monster database, in which making any progressive change gets exponentially more difficult due to lack of scalability and understanding of the model.

Using a simple example, a database to store user reviews of their favourite records, allow me to highlight a few really simple tips which could be your saviour if your database grows to become a monster.

Hot Tip 1 - Do Your Documentation

Most programmers and I.T. specialists simply do not like documenting what they do and managers tend to have to provoke them with red-hot irons for them to do so. I’ve been guilty of this on many occasions and I’ve had to pay dearly for this by spending vast amounts of time to re-analyse the workings of the database in order to make a change or to educate someone else so that they can understand the database and effect any changes as required.

Every object in your database (tables, views, sprocs, etc) and every field in every table should have a detailed description. This will form a part of your *data dictionary*. If your database supports it, I strongly recommend filling in any metadata fields against your database objects and fields as there are a number of very useful utilities on the market which can create your data dictionary for you simply by referencing these metadata fields. If your database does not support this, then you may need to create another table in your database to hold this information.

For example, in the table below, I have added three extended property tags explaining what the table is, who created it and when.

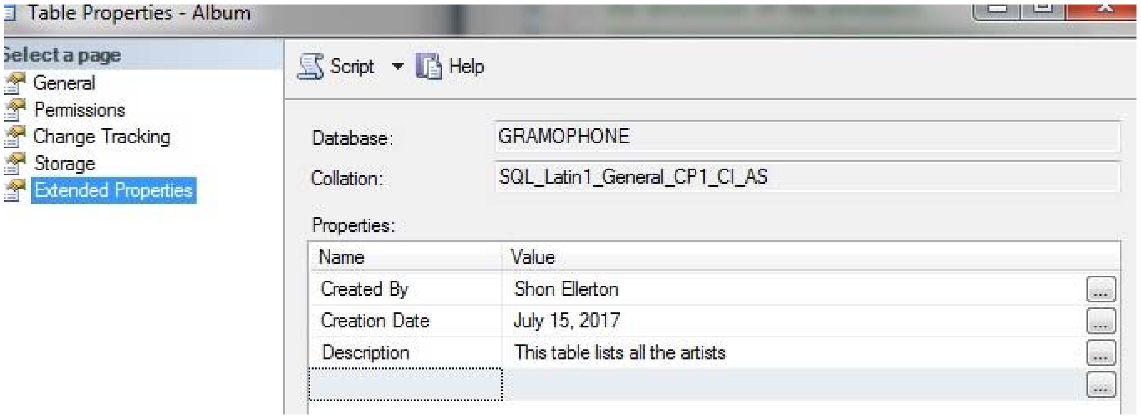


Table Properties - Album

Select a page: General, Permissions, Change Tracking, Storage, Extended Properties

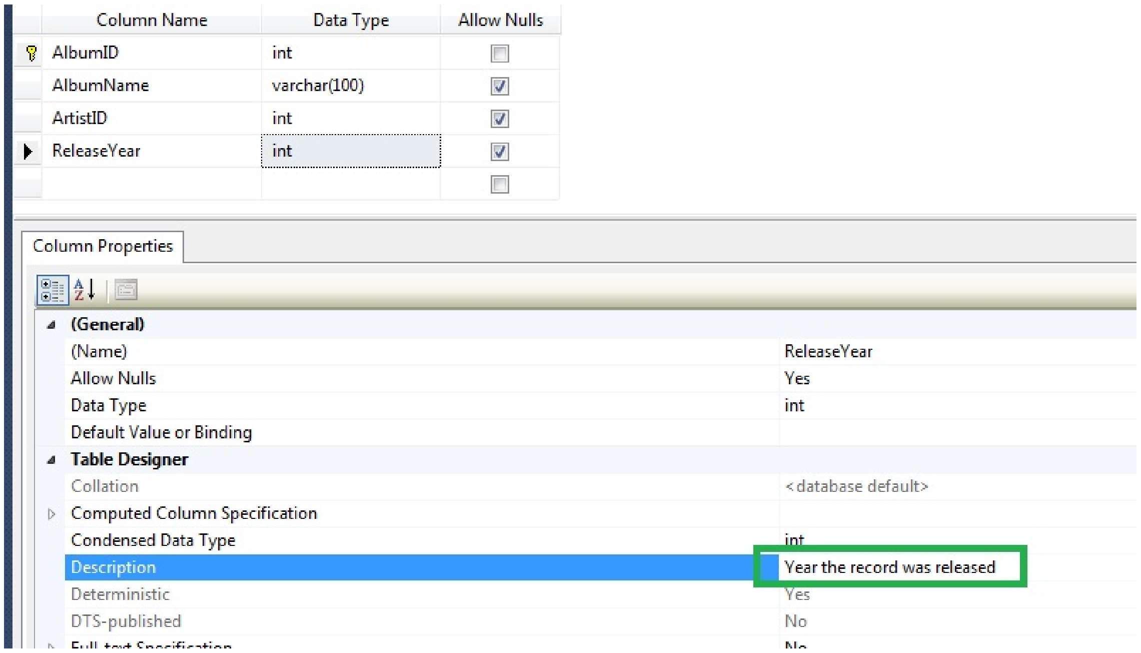
Database: GRAMOPHONE

Collation: SQL_Latin1_General_CP1_CI_AS

Properties:

Name	Value
Created By	Shon Ellerton
Creation Date	July 15, 2017
Description	This table lists all the artists

Adding detailed information for each field is important as well. Although it is self-explanatory in the below example, a description has been added to the field. You may be surprised how quickly you can forget what a field does once your database becomes more complex.

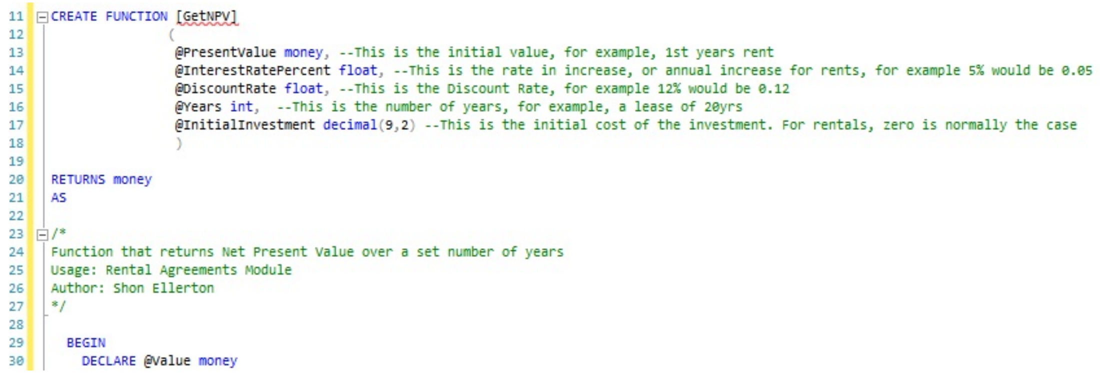


Column Name	Data Type	Allow Nulls
AlbumID	int	☐
AlbumName	varchar(100)	✓
ArtistID	int	✓
ReleaseYear	int	✓

Column Properties

(General)	
(Name)	ReleaseYear
Allow Nulls	Yes
Data Type	int
Default Value or Binding	
Table Designer	
Collation	<database default>
Computed Column Specification	
Condensed Data Type	int
Description	Year the record was released
Deterministic	Yes
DTS-published	No

As with programming languages, SQL code in your database, particularly with stored procedures and functions, should have plenty of commentary to ensure that you can return back to the code with confidence as to why it was put there in the first place. The example below shows some commentary (in green) for a user-defined function I created to calculate NPV (net present value). If anyone opened this function, they should be able to dissect it without any problem. Note that it may be useful for others if the author's name is included.



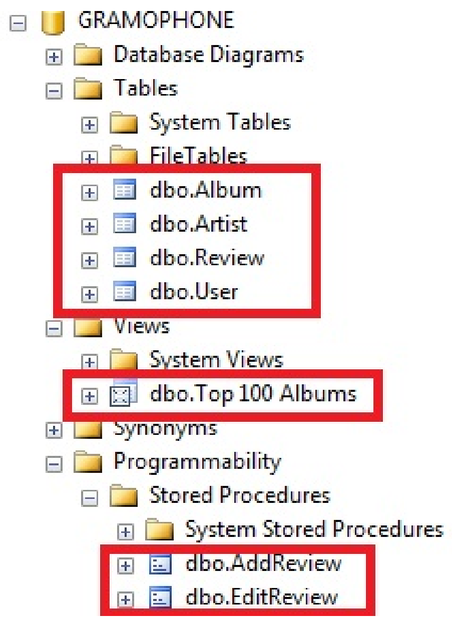
```
CREATE FUNCTION [GetNPV]
                (
                @PresentValue money, --This is the initial value, for example, 1st years rent
                @InterestRatePercent float, --This is the rate in increase, or annual increase for rents, for example 5% would be 0.05
                @DiscountRate float, --This is the Discount Rate, for example 12% would be 0.12
                @Years int,  --This is the number of years, for example, a lease of 20yrs
                @InitialInvestment decimal(9,2) --This is the initial cost of the investment. For rentals, zero is normally the case
                )

RETURNS money
AS

/*
Function that returns Net Present Value over a set number of years
Usage: Rental Agreements Module
Author: Shon Ellerton
*/

  BEGIN
    DECLARE @Value money
```

This basic documentation forms only a part of what is required for a complete work management system. You may need to document other areas including: process, data import and export flows, security and step-by-step guides for users along with a high-level document for senior management and non-technical readers.

Hot 2 - Provide Good Names

Naming your database objects and fields correctly is the easiest tip to follow *if* you are the only person creating these objects and fields. I really suggest setting up some rules on how objects and fields are named in the database. Many would think that this exercise is trivial and trite bordering on being excessive compulsive; however, it really makes sense to get this right as I will explain.

The below example shows, what I believe to be, a poorly named collection of objects in the database. They are boxed in red.



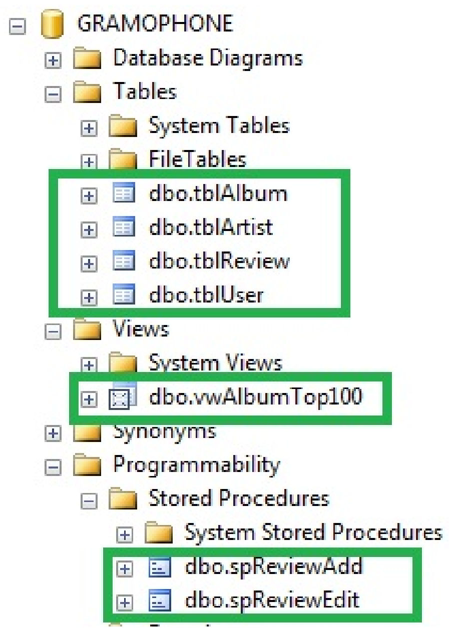
So what is wrong with them?

The name of the object does not tell me what kind of object they are. For example, **Album** *could* refer to a view or other object if one came across this in an SQL query without the user knowing what the database schema looks like.

Top 100 Albums has embedded spaces, which is *never* a good idea. Moreover, the word, *album*, should be first rather than *top*.

AddReview and **EditReview** are, gauging from the name, likely to be stored procedures; however, it is not absolutely certain this is so.

Now I'm not saying that you follow my suggestion; however, a consistent approach should be adopted throughout the life of the database. The example below is a way in which, I personally, like to adopt for database naming conventions. All others working on the database need to adopt one agreed approach. Note the changes in the green boxes.



The naming convention above is far more organised then the previous example. Let me tell you now that I have worked with databases with hundreds of objects, and if you do not have an *agreed* uniform approach to object naming, it becomes very difficult to find objects in your database.

Each object is prefixed to identify whether it is a **table,** a **view**, or a **stored procedure**.

To make the objects easier to search alphabetically:

Top 100 Albums becomes **vwAlbumTop100**.

AddReview becomes **spReviewAdd**.

EditReview becomes **spReviewEdit**.

Yes, naming may seem trivial, but it is incredibly easy to get this right.

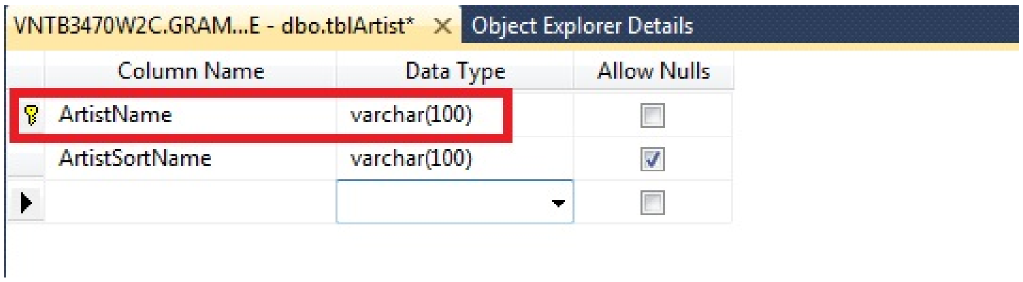
Some further tips include:

1) Taking care not to make spelling mistakes;
2) Avoid special characters and spaces;
3) Be consistent with the plural or singular form. Use either one or the other. I stick with the singular form personally.

Hot Tip 3 - Don’t Forget Your Primary Keys

This is an area where so much can go wrong. In more than 95% of all the tables I have designed, I have included only *one* integer-based primary key.

The below is a bad example of a primary key in which the artist’s name is being used.

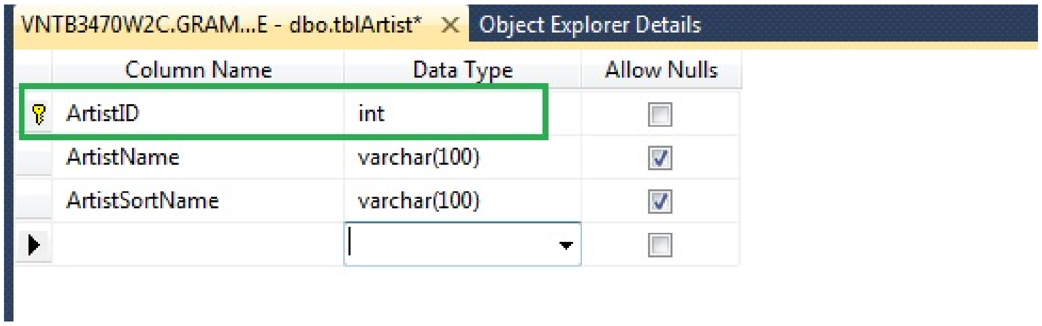


Column Name	Data Type	Allow Nulls
ArtistName	varchar(100)	☐
ArtistSortName	varchar(100)	✓
		☐

This is bad in 3 major ways:

1) There may be more than one artist with the same name;
2) Changing the name of the artist will most likely ‘break’ your table structure;
3) Writing SQL using integer-based primary keys is far easier, especially when doing multiple joins.

Do this instead. This keeps the **ArtistID** separate from the artist's name.



Column Name	Data Type	Allow Nulls
ArtistID	int	☐
ArtistName	varchar(100)	✓
ArtistSortName	varchar(100)	✓
		☐

Should I use autoincrement for my primary key?

For those not familiar, autoincrement is when the database automatically assigns a new integer primary key for you when a new record is inserted.

My personal view is this.

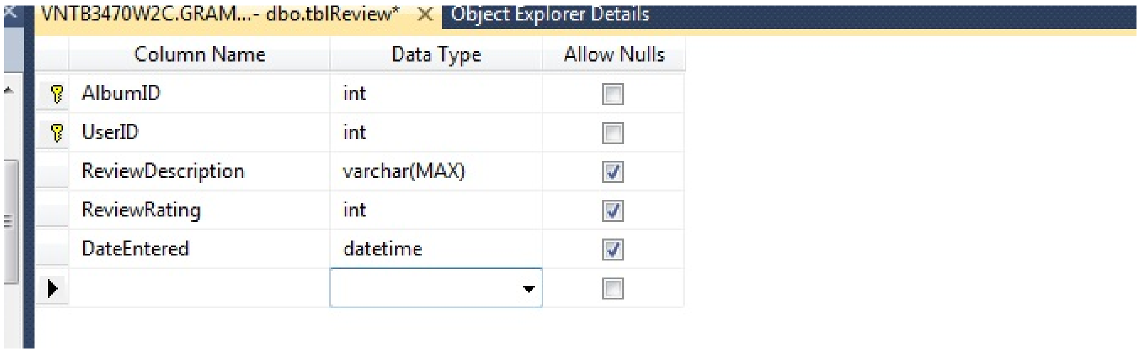
If the table is small or meant to be a lookup table. For example, a table which has **categories** of records or a table which lists the states in the country I live in, I prefer *not* to use autoincrement. However, for large tables, for example, reviews from all users being added to the database on a regular basis, I would suggest using autoincrement.

Should I have the primary key go *UP* in order or *DOWN* in order?

Again, this is personal preference, but for large tables which have continuous new inserts (for example, log tables), I *always* prefer going ***down*** in order. In other words, I am far more interested to see the most recent entries first if doing a standard **select** query. If you have a very large table and wish to change the primary key order, do this on a *copy* of the table as this procedure could take some time to complete and you do not want to be in the embarrassing situation of locking up your working table.

Should I have a separate primary key for a junction table?

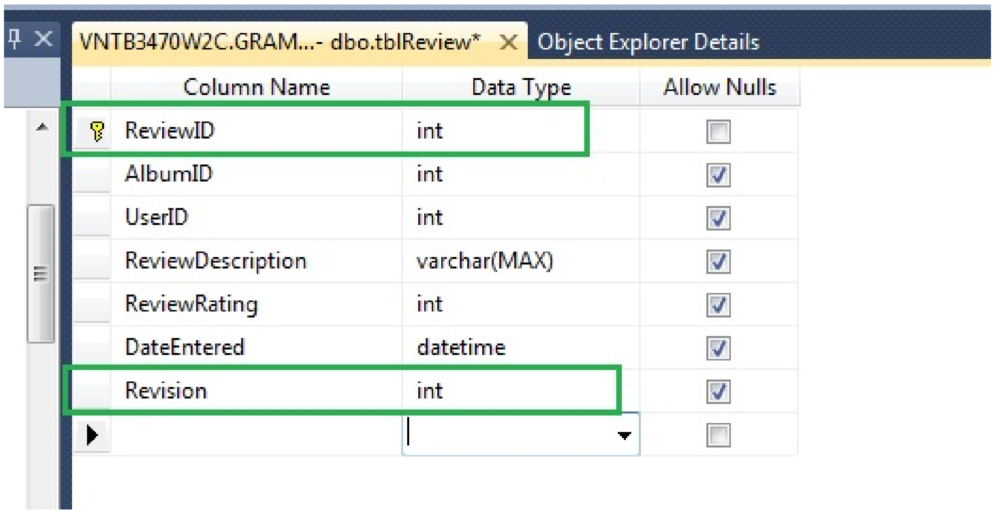
In my opinion, yes! It does not do any harm in doing so. Take for example a table of user reviews. I *could* simply assign **UserID** and **AlbumID** as a *combination primary key* as can be seem below.



VNTB3470W2C.GRAM...- dbo.tblReview* Object Explorer Details

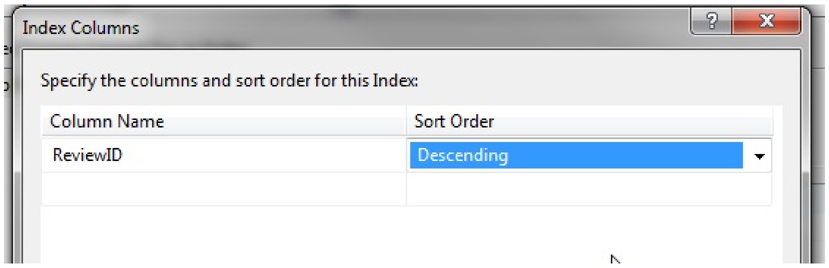
Column Name	Data Type	Allow Nulls
AlbumID	int	☐
UserID	int	☐
ReviewDescription	varchar(MAX)	✓
ReviewRating	int	✓
DateEntered	datetime	✓
		☐

However, what happens when that user wants to add a *revised* review? You can only write over the old entry as adding another review for the same album will not be allowed. To solve this problem, here is the new and improved version with an *autoincrement descending* primary key.



VNTB3470W2C.GRAM...- dbo.tblReview* Object Explorer Details

Column Name	Data Type	Allow Nulls
ReviewID	int	☐
AlbumID	int	✓
UserID	int	✓
ReviewDescription	varchar(MAX)	✓
ReviewRating	int	✓
DateEntered	datetime	✓
Revision	int	✓
		☐



Index Columns

Specify the columns and sort order for this Index:

Column Name	Sort Order
ReviewID	Descending

Do all tables need primary keys?

There are circumstances where primary keys are not required; for example, in *staging or temporary tables*. For example, if you have developed an automated import process from an external source, you will need to extract and dump the data into a staging table in which the data will undergo data-cleansing before being finally merged into a structured table with a primary key.

Should I start with the number, 1, as the first record?

It is not mandatory by any means, but it is generally good practice, if possible, to keep the string length of the integer consistent. You may note that this is common practice in consumer billing for example. If you envisage 100,000 records, why not start (or *seed*) with 100001 as your first record. This will allow plenty of leeway up to 999999 and you maintain the same number of characters. If you were to start with 1 and you wanted to keep the same length and show as 000001, this would require converting the integer to a string – an extra unnecessary step.

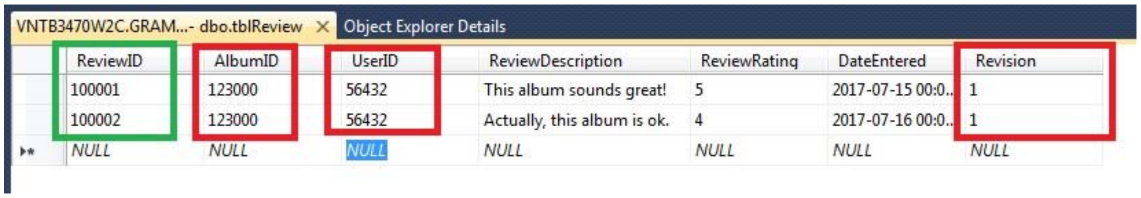
As you can see, ensuring that you have good primary keys in your tables is an important foundation to building a solid database schema.

Hot Tip 4 - Prevent Unwanted Duplications

Just because you have a primary key, it does not necessarily mean that this will prevent possible unwanted duplications in your tables.

For example, let's go back to the user review table.

When the user writes the revised review, you would not want to have the user interface or application assign the same revision as below.



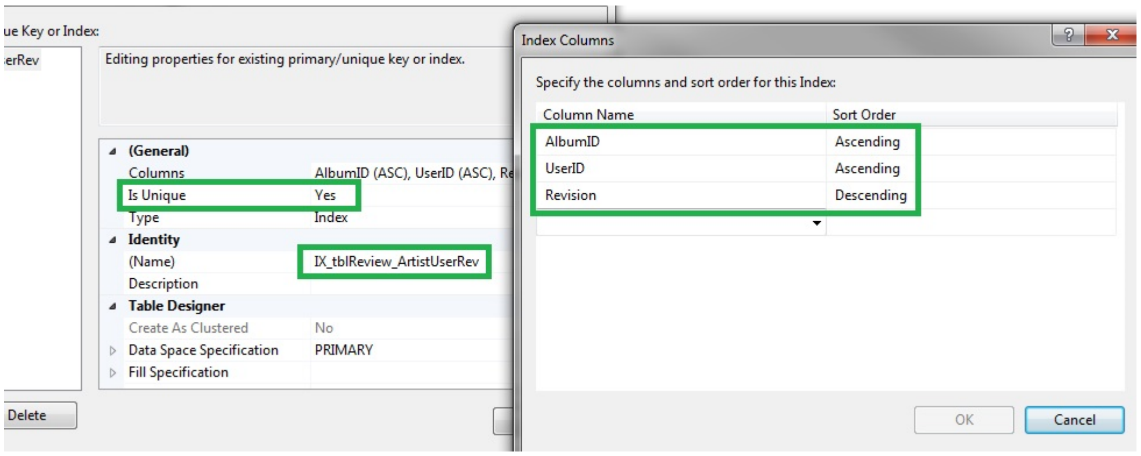
VNTB3470W2C.GRAM...- dbo.tblReview × Object Explorer Details

	ReviewID	AlbumID	UserID	ReviewDescription	ReviewRating	DateEntered	Revision
	100001	123000	56432	This album sounds great!	5	2017-07-15 00:0...	1
	100002	123000	56432	Actually, this album is ok.	4	2017-07-16 00:0...	1
▸*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

This is probably the most common mistake made; forgetting to add unique indexes when required.

The **ReviewID** is the primary key which is all good; however, you would not want to have the **AlbumID, UserID** and **Revision** be duplicated.

Therefore, it is important to add a new unique index to this table. I created a *unique* index and specified the **revision** field to be *descending* rather than ascending.


ue Key or Index:

erRev

Editing properties for existing primary/unique key or index.

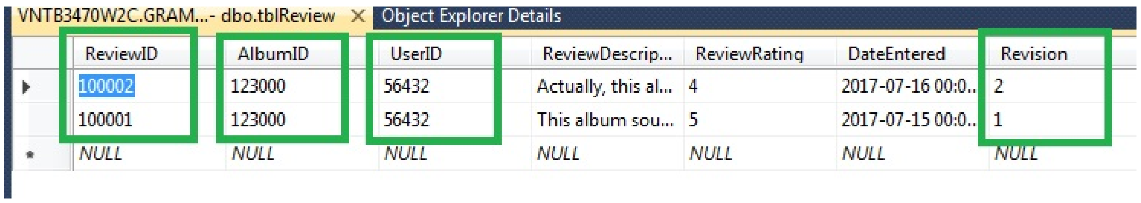
(General)	
Columns	AlbumID (ASC), UserID (ASC), Re
Is Unique	Yes
Type	Index
Identity	
(Name)	IX_tblReview_ArtistUserRev
Description	
Table Designer	
Create As Clustered	No
Data Space Specification	PRIMARY
Fill Specification	

Index Columns

Specify the columns and sort order for this Index:

Column Name	Sort Order
AlbumID	Ascending
UserID	Ascending
Revision	Descending

Now the results are looking considerably better!


VNTB3470W2C.GRAM...- dbo.tblReview   Object Explorer Details

ReviewID	AlbumID	UserID	ReviewDescrip...	ReviewRating	DateEntered	Revision
100002	123000	56432	Actually, this al...	4	2017-07-16 00:0...	2
100001	123000	56432	This album sou...	5	2017-07-15 00:0...	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL

The art of indexing is a science on its own, but this is where indexing is extremely useful to avoid those embarrassing unwanted duplications. Make sure to specify *unique* for the index of course!

Hot Tip 5 - Build Relationships

Doesn’t this sound like your typical LinkedIn article? ‘Build Relationships’… but not in this context!

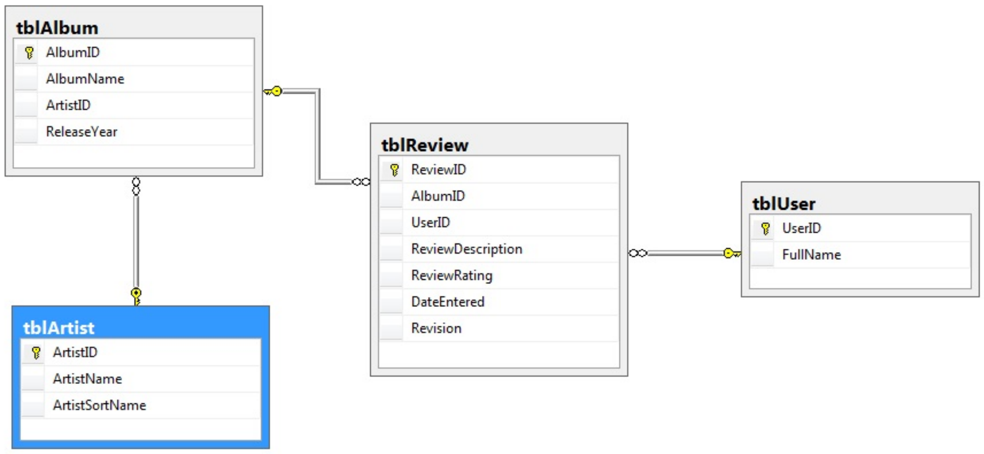
Relationships. So easy to forget! I’ve done it myself once or twice.

I’ll make this one short and sweet.

Don’t forget to add your relationships between tables and don’t forget to enforce *referential integrity*.

Once other thing, be very careful with the *cascade delete* option. This is when, if a record is deleted from a table, it will delete underlying records in dependent tables. It is safer to do this in code or through a stored procedure. However, depending on the circumstances, the underlying information may not be of concern if it is removed.

If you are using integer primary keys, I see no real reason to use the *cascade update* option.



And Finally…

To recapitulate:

1) Do your documentation;
2) Provide good names;
3) Don't forget your primary keys;
4) Prevent unwanted duplications;
5) Build relationships.

I'm sure there are plenty of other hot tips and I would be grateful to receive any feedback along with more useful tips.

Thanks for reading!