

Schema for a multilanguage database

Asked 12 years, 10 months ago Active 1 year ago Viewed 103k times



260



154



I'm developing a multilanguage software. As far as the application code goes, localizability is not an issue. We can use language specific resources and have all kinds of tools that work well with them.

But what is the best approach in defining a multilanguage database schema? Let's say we have a lot of tables (100 or more), and each table can have multiple columns that can be localized (most of nvarchar columns should be localizable). For instance one of the tables might hold product information:

```
CREATE TABLE T_PRODUCT (  
    NAME          NVARCHAR(50),  
    DESCRIPTION    NTEXT,  
    PRICE          NUMBER(18, 2)  
)
```

I can think of three approaches to support multilingual text in NAME and DESCRIPTION columns:

1. Separate column for each language

When we add a new language to the system, we must create additional columns to store the translated text, like this:

```
CREATE TABLE T_PRODUCT (  
    NAME_EN        NVARCHAR(50),  
    NAME_DE        NVARCHAR(50),  
    NAME_SP        NVARCHAR(50),  
    DESCRIPTION_EN  NTEXT,  
    DESCRIPTION_DE  NTEXT,  
    DESCRIPTION_SP  NTEXT,  
    PRICE          NUMBER(18,2)  
)
```

2. Translation table with columns for each language

Instead of storing translated text, only a foreign key to the translations table is stored. The translations table contains a column for each language.

```
CREATE TABLE T_PRODUCT (  
  NAME_FK          int,  
  DESCRIPTION_FK   int,  
  PRICE            NUMBER(18, 2)  
)  
  
CREATE TABLE T_TRANSLATION (  
  TRANSLATION_ID,  
  TEXT_EN NTEXT,  
  TEXT_DE NTEXT,  
  TEXT_SP NTEXT  
)
```

3. Translation tables with rows for each language

Instead of storing translated text, only a foreign key to the translations table is stored. The translations table contains only a key, and a separate table contains a row for each translation to a language.

```
CREATE TABLE T_PRODUCT (  
  NAME_FK          int,  
  DESCRIPTION_FK   int,  
  PRICE            NUMBER(18, 2)  
)  
  
CREATE TABLE T_TRANSLATION (  
  TRANSLATION_ID  
)  
  
CREATE TABLE T_TRANSLATION_ENTRY (  
  TRANSLATION_FK,  
  LANGUAGE_FK,  
  TRANSLATED_TEXT NTEXT  
)  
  
CREATE TABLE T_TRANSLATION_LANGUAGE (  
  LANGUAGE_ID,  
  LANGUAGE_CODE CHAR(2)  
)
```

There are pros and cons to each solution, and I would like to know what are your experiences with these approaches, what do you recommend and how would you go about designing a multilanguage database schema.

Share Edit Follow

edited Jul 15 '14 at 8:06



user247702

22.3k

13

105

146

asked Nov 25 '08 at 9:13



qbeuek

3,995

4

18

17

14 Check codeproject.com/KB/aspnet/... stackoverflow.com/questions/3077305/... stackoverflow.com/questions/929410/... – Frosty Z Jan 20 '11 at 10:40



3 You can check this link: gsdesign.ro/blog/multilanguage-database-design-approach although reading the comments is very helpful – Fareed Alnamrouti Jan 18 '12 at 13:52

4 LANGUAGE_CODE are natural key, avoid LANGUAGE_ID . – gavenkoa Jun 4 '14 at 8:15

1 I already seen/used the 2. and 3., I don't recommend them, you easily end up with orphaned rows. @SunWiKung design looks better IMO. – Guillaume86 Dec 23 '14 at 13:34

4 I prefer SunWuKungs design, which coincidentally is what we have implemented. However, you need to consider collations. In Sql Server at least, each column has a collation property, which determines things like case sensitivity, equivalence (or not) of accented characters, and other language-specific considerations. Whether you use language-specific collations or not depends on your overall application design, but if you get it wrong, it'll be hard to change later. If you need language-specific collations, then you'll need a column per language, not a row per language. – Elroy Flynn Feb 10 '15 at 20:45

12 Answers

Active

Oldest

Votes



What do you think about having a related translation table for each translatable table?

125



```
CREATE TABLE T_PRODUCT (pr_id int, PRICE NUMBER(18, 2))
```

```
CREATE TABLE T_PRODUCT_tr (pr_id INT FK, languagecode varchar, pr_name text, pr_descr text)
```



This way if you have multiple translatable column it would only require a single join to get it + since you are not autogenerating a translationid it may be easier to import items together with their related translations.

The negative side of this is that if you have a complex language fallback mechanism you may need to implement that for each translation table - if you are relying on some stored procedure to do that. If you do that from the app this will probably not be a

problem.

Let me know what you think - I am also about to make a decision on this for our next application. So far we have used your 3rd type.

Share Edit Follow

edited Jun 20 '20 at 9:12

answered Nov 27 '08 at 10:02



Community ♦

1 1

SunWuKung

-
- 2 This option is similar to my option nr 1 but better. It is still hard to maintain and requires creating new tables for new languages, so I'd be reluctant to implement it. – [qbeuek](#) Nov 27 '08 at 10:22
-
- 30 it doesn't require a new table for a new language - you simply add a new row to the appropriate _tr table with your new language, you only need to create a new _tr table if you create a new translatable table – [SunWuKung](#) Nov 27 '08 at 12:37
-
- 3 i beleive that this is a good method. other methods require tons of left joins and when you are joining multiple tables that each of them have translation like 3 level deep, and each one has 3 fields you need 3*3 9 left joins only for translations.. other wise 3. Also it is easier to add constraints etc and i beleive searching is more resonable. – [GorillaApe](#) May 1 '12 at 10:08
-
- 1 When T_PRODUCT has 1 million rows, T_PRODUCT_tr would have 2 million.Would it reduce sql efficiency much? – [Mithril](#) Feb 14 '14 at 3:22
-
- 1 @Mithril Either way you have 2 million rows. At least you don't need joins with this method. – [David D](#) Sep 8 '14 at 11:50 ✎
-



This is an interesting issue, so let's necromance.

66

Let's start by the problems of method 1:

Problem: You're denormalizing to save speed.



In SQL (except PostGreSQL with hstore), you can't pass a parameter language, and say:



```
SELECT ['DESCRIPTION_' + @in_language] FROM T_Products
```

So you have to do this:

```
SELECT
    Product_UID
,
    CASE @in_language
        WHEN 'DE' THEN DESCRIPTION_DE
```

```

        WHEN 'SP' THEN DESCRIPTION_SP
        ELSE DESCRIPTION_EN
    END AS Text
FROM T_Products

```

Which means you have to alter ALL your queries if you add a new language. This naturally leads to using "dynamic SQL", so you don't have to alter all your queries.

This usually results in something like this (and it can't be used in views or table-valued functions by the way, which really is a problem if you actually need to filter the reporting date)

```

CREATE PROCEDURE [dbo].[sp_RPT_DATA_BadExample]
    @in_mandant varchar(3)
    ,@in_language varchar(2)
    ,@in_building varchar(36)
    ,@in_wing varchar(36)
    ,@in_reportingdate varchar(50)
AS
BEGIN
    DECLARE @sql varchar(MAX), @reportingdate datetime

    -- Abrunden des Eingabedatums auf 00:00:00 Uhr
    SET @reportingdate = CONVERT( datetime, @in_reportingdate)
    SET @reportingdate = CAST(FLOOR(CAST(@reportingdate AS float)) AS datetime)
    SET @in_reportingdate = CONVERT(varchar(50), @reportingdate)

    SET NOCOUNT ON;

    SET @sql='SELECT
        Building_Nr AS RPT_Building_Number
        ,Building_Name AS RPT_Building_Name
        ,FloorType_Lang_ ' + @in_language + ' AS RPT_FloorType
        ,Wing_No AS RPT_Wing_Number
        ,Wing_Name AS RPT_Wing_Name
        ,Room_No AS RPT_Room_Number
        ,Room_Name AS RPT_Room_Name
    FROM V_Whatever
    WHERE SO_MDT_ID = ''' + @in_mandant + '''

    AND
    (
        ''' + @in_reportingdate + ''' BETWEEN CAST(FLOOR(CAST(Room_DateFrom AS float)) AS datetime) AND Room_DateTo
        OR Room_DateFrom IS NULL
        OR Room_DateTo IS NULL
    )

```

```

    )
    ,
    IF @in_building <> '00000000-0000-0000-0000-000000000000' SET @sql=@sql + 'AND (Building_UID = ''' + @in_building + ''')
```

The problem with this is

a) Date-formatting is very language-specific, so you get a problem there, if you don't input in ISO format (which the average garden-variety programmer usually doesn't do, and in case of a report the user sure as hell won't do for you, even if explicitly instructed to do so).

and

b) **most significantly**, you **loose any kind of syntax checking**. If <insert name of your "favourite" person here> alters the schema because suddenly the requirements for wing change, and a new table is created, the old one left but the reference field renamed, you don't get any kind of warning. A report even works **when you run it without selecting the wing parameter** (==> guid.empty). But suddenly, when an actual user actually selects a wing ==> **boom. This method completely breaks any kind of testing.**

Method 2:

In a nutshell: "Great" idea (warning - sarcasm), let's combine the disadvantages of method 3 (slow speed when many entries) with the rather horrible disadvantages of method 1.

The only advantage of this method is that you keep all translation in one table, and therefore make maintenance simple. However, the same thing can be achieved with method 1 and a dynamic SQL stored procedure, and a (possibly temporary) table containing the translations, and the name of the target table (and is quite simple assuming you named all your text-fields the same).

Method 3:

One table for all translations: Disadvantage: You have to store n Foreign Keys in the products table for n fields you want to translate. Therefore, you have to do n joins for n fields. When the translation table is global, it has many entries, and joins become slow. Also, you always have to join the T_TRANSLATION table n times for n fields. This is quite an overhead. Now, what do you do when you must accommodate custom translations per customer ? You'll have to add another 2x n joins onto an additional table. If you have to join , say 10 tables, with 2x2xn = 4n additional joins, what a mess ! Also, this design makes it possible to use the same translation with 2 tables. If I change the item name in one table, do I really want to change an entry in another table as well EVERY SINGLE TIME ?

Plus you can't delete and re-insert the table anymore, because there are now foreign keys IN THE PRODUCT TABLE(s)... you can of course omit setting the FKs, and then <insert name of your "favourite" person here> can delete the table, and re-insert all entries with

newid() [or by specifying the id in the insert, but having **identity-insert OFF**], and that would (and will) lead to data-garbage (and null-reference exceptions) really soon.

Method 4 (not listed): Storing all the languages in a XML field in the database. e.g

```
-- CREATE TABLE MyTable(myfilename nvarchar(100) NULL, filemeta xml NULL )

;WITH CTE AS
(
    -- INSERT INTO MyTable(myfilename, filemeta)
    SELECT
        'test.mp3' AS myfilename
        --, CONVERT(XML, N'<?xml version="1.0" encoding="utf-16" standalone="yes"?><body>Hello</body>', 2)
        --, CONVERT(XML, N'<?xml version="1.0" encoding="utf-16" standalone="yes"?><body><de>Hello</de></body>', 2)
        , CONVERT(XML
            , N'<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<lang>
    <de>Deutsch</de>
    <fr>Français</fr>
    <it>Ital&amp;iano</it>
    <en>English</en>
</lang>
    ,
    , 2
    ) AS filemeta
)

SELECT
    myfilename
    , filemeta
    --, filemeta.value('body', 'nvarchar')
    --, filemeta.value('.', 'nvarchar(MAX)')

    , filemeta.value('(/lang//de/node())[1]', 'nvarchar(MAX)') AS DE
    , filemeta.value('(/lang//fr/node())[1]', 'nvarchar(MAX)') AS FR
    , filemeta.value('(/lang//it/node())[1]', 'nvarchar(MAX)') AS IT
    , filemeta.value('(/lang//en/node())[1]', 'nvarchar(MAX)') AS EN
FROM CTE
```

Then you can get the value by XPath-Query in SQL, where you can put the string-variable in

```
filemeta.value('(/lang//' + @in_language + '/node())[1]', 'nvarchar(MAX)') AS bla
```

And you can update the value like this:

```
UPDATE YOUR_TABLE
SET YOUR_XML_FIELD_NAME.modify('replace value of (/lang/de/text())[1] with "&quot;I am a ''value &quot;";')
WHERE id = 1
```

Where you can replace `/lang/de/...` with `'.../' + @in_language + '/...'`

Kind of like the PostgreSQL hstore, except that due to the overhead of parsing XML (instead of reading an entry from an associative array in PG hstore) it becomes far too slow plus the xml encoding makes it too painful to be useful.

Method 5 (as recommended by SunWuKung, the one you should choose): One translation table for each "Product" table. That means one row per language, and several "text" fields, so it requires only ONE (left) join on N fields. Then you can easily add a default-field in the "Product"-table, you can easily delete and re-insert the translation table, and you can create a second table for custom-translations (on demand), which you can also delete and re-insert), and you still have all the foreign keys.

Let's make an example to see this WORKS:

First, create the tables:

```
CREATE TABLE dbo.T_Languages
(
    Lang_ID int NOT NULL
    ,Lang_NativeName national character varying(200) NULL
    ,Lang_EnglishName national character varying(200) NULL
    ,Lang_ISO_TwoLetterName character varying(10) NULL
    ,CONSTRAINT PK_T_Languages PRIMARY KEY ( Lang_ID )
);

GO
```

```
CREATE TABLE dbo.T_Products
(
    PROD_Id int NOT NULL
    ,PROD_InternalName national character varying(255) NULL
```



```

        ,CONSTRAINT PK_T_Products PRIMARY KEY ( PROD_Id )
    );

GO

CREATE TABLE dbo.T_Products_i18n
(
    PROD_i18n_PROD_Id int NOT NULL
    ,PROD_i18n_Lang_Id int NOT NULL
    ,PROD_i18n_Text national character varying(200) NULL
    ,CONSTRAINT PK_T_Products_i18n PRIMARY KEY (PROD_i18n_PROD_Id, PROD_i18n_Lang_Id)
);

GO

-- ALTER TABLE dbo.T_Products_i18n WITH NOCHECK ADD CONSTRAINT FK_T_Products_i18n_T_Products FOREIGN KEY(PROD_i18n_PROD_Id)
ALTER TABLE dbo.T_Products_i18n
    ADD CONSTRAINT FK_T_Products_i18n_T_Products
    FOREIGN KEY (PROD_i18n_PROD_Id)

```

Then fill in the data

```

DELETE FROM T_Languages;
INSERT INTO T_Languages (Lang_ID, Lang_NativeName, Lang_EnglishName, Lang_ISO_TwoLetterName) VALUES (1, N'English', N'English', N'EN');
INSERT INTO T_Languages (Lang_ID, Lang_NativeName, Lang_EnglishName, Lang_ISO_TwoLetterName) VALUES (2, N'Deutsch', N'German', N'DE');
INSERT INTO T_Languages (Lang_ID, Lang_NativeName, Lang_EnglishName, Lang_ISO_TwoLetterName) VALUES (3, N'Français', N'French', N'FR');
INSERT INTO T_Languages (Lang_ID, Lang_NativeName, Lang_EnglishName, Lang_ISO_TwoLetterName) VALUES (4, N'Italiano', N'Italian', N'IT');
INSERT INTO T_Languages (Lang_ID, Lang_NativeName, Lang_EnglishName, Lang_ISO_TwoLetterName) VALUES (5, N'Russki', N-Russian', N'RU');
INSERT INTO T_Languages (Lang_ID, Lang_NativeName, Lang_EnglishName, Lang_ISO_TwoLetterName) VALUES (6, N'Zhungwen', N'Chinese', N'ZH');

DELETE FROM T_Products;
INSERT INTO T_Products (PROD_Id, PROD_InternalName) VALUES (1, N'Orange Juice');
INSERT INTO T_Products (PROD_Id, PROD_InternalName) VALUES (2, N'Apple Juice');
INSERT INTO T_Products (PROD_Id, PROD_InternalName) VALUES (3, N'Banana Juice');
INSERT INTO T_Products (PROD_Id, PROD_InternalName) VALUES (4, N'Tomato Juice');
INSERT INTO T_Products (PROD_Id, PROD_InternalName) VALUES (5, N'Generic Fruit Juice');

DELETE FROM T_Products_i18n;
INSERT INTO T_Products_i18n (PROD_i18n_PROD_Id, PROD_i18n_Lang_Id, PROD_i18n_Text) VALUES (1, 1, N'Orange Juice');
INSERT INTO T_Products_i18n (PROD_i18n_PROD_Id, PROD_i18n_Lang_Id, PROD_i18n_Text) VALUES (1, 2, N'Orangensaft');

```

```

INSERT INTO T_Products_i18n (PROD_i18n_PROD_Id, PROD_i18n_Lang_Id, PROD_i18n_Text) VALUES (1, 3, N'Jus d''Orange');
INSERT INTO T_Products_i18n (PROD_i18n_PROD_Id, PROD_i18n_Lang_Id, PROD_i18n_Text) VALUES (1, 4, N'Succo d''arancia');
INSERT INTO T_Products_i18n (PROD_i18n_PROD_Id, PROD_i18n_Lang_Id, PROD_i18n_Text) VALUES (2, 1, N'Apple Juice');
INSERT INTO T_Products_i18n (PROD_i18n_PROD_Id, PROD_i18n_Lang_Id, PROD_i18n_Text) VALUES (2, 2, N'Apfelsaft');

DELETE FROM T_Products_i18n_Cust;
INSERT INTO T_Products_i18n_Cust (PROD_i18n_Cust_PROD_Id, PROD_i18n_Cust_Lang_Id, PROD_i18n_Cust_Text) VALUES (1, 2,
N'Orangäsafft'); -- Swiss German, if you wonder

```

And then query the data:

```

DECLARE @__in_lang_id int
SET @__in_lang_id = (
    SELECT Lang_ID
    FROM T_Languages
    WHERE Lang_ISO_TwoLetterName = 'DE'
)

SELECT
    PROD_Id
    ,PROD_InternalName -- Default Fallback field (internal name/one language only setup), just in ResultSet for demo-purposes
    ,PROD_i18n_Text -- Translation text, just in ResultSet for demo-purposes
    ,PROD_i18n_Cust_Text -- Custom Translations (e.g. per customer) Just in ResultSet for demo-purposes
    ,COALESCE(PROD_i18n_Cust_Text, PROD_i18n_Text, PROD_InternalName) AS DisplayText -- What we actually want to show
FROM T_Products

LEFT JOIN T_Products_i18n
    ON PROD_i18n_PROD_Id = T_Products.PROD_Id
    AND PROD_i18n_Lang_Id = @__in_lang_id

LEFT JOIN T_Products_i18n_Cust
    ON PROD_i18n_Cust_PROD_Id = T_Products.PROD_Id
    AND PROD_i18n_Cust_Lang_Id = @__in_lang_id

```

If you're lazy, then you can also use the ISO-TwoLetterName ('DE', 'EN', etc.) as primary-key of the language table, then you don't have to lookup the language id. But if you do so, you maybe want to use the [IETF-language tag](#) instead, which is better, because you get de-CH and de-DE, which is really not the same ortography-wise (double s instead of ß everywhere), although it's the same base-language. That just as a tiny little detail that may be important to you, especially considering that en-US and en-GB/en-CA/en-AU or fr-FR/fr-CA has similar issues.

Quote: we don't need it, we only do our software in English.

Answer: Yes - but which one ??

Anyway, if you use an integer ID, you're flexible, and can change your method at any later time.
And you should use that integer, because there's nothing more annoying, destructive and troublesome than a botched Db design.

See also [RFC 5646](#), [ISO 639-2](#),

And, if you're still saying "we" *only* make our application for "only **one** culture" (like en-US usually)- therefore I don't need that extra integer, this would be a good time and place to mention the [IANA language tags](#), wouldn't it ?

Because they go like this:

```
de-DE-1901  
de-DE-1996
```

and

```
de-CH-1901  
de-CH-1996
```

(there was an orthography reform in 1996...) Try finding a word in a dictionary if it is misspelled; this becomes very important in applications dealing with legal and public service portals.

More importantly, there are regions that are changing from cyrillic to latin alphabets, which may just be more troublesome than the superficial nuisance of some obscure orthography reform, which is why this might be an important consideration too, depending on which country you live in. One way or the other, it's better to have that integer in there, just in case...

Edit:

And by adding `ON DELETE CASCADE` after

```
REFERENCES dbo.T_Products( PROD_Id )
```

you can simply say: `DELETE FROM T_Products` , and get no foreign key violation.

As for collation, I'd do it like this:

- A) Have your own DAL
- B) Save the desired collation name in the language table

You might want to put the collations in their own table, e.g.:

```
SELECT * FROM sys.fn_helpcollations()  
WHERE description LIKE '%insensitive'  
AND name LIKE '%german%'
```

C) Have the collation name available in your `auth.user.language` information

D) Write your SQL like this:

```
SELECT  
    COALESCE(GRP_Name_i18n_cust, GRP_Name_i18n, GRP_Name) AS GroupName  
FROM T_Groups  
  
ORDER BY GroupName COLLATE {#COLLATION}
```

E) Then, you can do this in your DAL:

```
cmd.CommandText = cmd.CommandText.Replace("{#COLLATION}", auth.user.language.collation)
```

Which will then give you this perfectly composed SQL-Query

```
SELECT  
    COALESCE(GRP_Name_i18n_cust, GRP_Name_i18n, GRP_Name) AS GroupName  
FROM T_Groups  
  
ORDER BY GroupName COLLATE German_PhoneBook_CI_AI
```

Share Edit Follow

edited May 20 '19 at 15:06

answered Sep 26 '14 at 9:28



Stefan Steiger

69.9k

63


344

414

Good detailed response, many thanks. But what do you think about the collation issues in the Method 5 solution. It seems this is not the best way when you needed to sort or to filter the translated text in the multilingual environment with different collations. And in such case the Method 2

(which you "ostracized" so quickly :)) could be a better option with slight modifications indicating target collation for each localized column.

– [Eugene Evdokimov](#) Aug 31 '15 at 12:06

- 2 @Eugene Evdokimov: Yes, but "ORDER BY" is always going to be a problem, because you can't specify it as a variable. My approach would be to save the collation name in the language table, and have this in the userinfo. Then, on each SQL-Statement you can say ORDER BY COLUMN_NAME {#collation}, and then you can do a replace in your dal (cmd.CommandText = cmd.CommandText.Replace("{#COLLATION}", auth.user.language.collation) . Alternatively, you can sort in your application code, e.g. using LINQ. This would also take some processing load off your database. For reports, the report sorts anyway. – [Stefan Steiger](#) Sep 3 '15 at 8:11 

o.o This must be the longest SO answer I've seen, and I saw people make whole programs in answers. You're good. – [Domino](#) Oct 7 '15 at 14:46

Can totally agree SunWuKung's solution is the best – [Domi](#) Aug 14 '18 at 8:00



The third option is the best, for a few reasons:

49



- Doesn't require altering the database schema for new languages (and thus limiting code changes)
- Doesn't require a lot of space for unimplemented languages or translations of a particular item
- Provides the most flexibility
- You don't end up with sparse tables
- You don't have to worry about null keys and checking that you're displaying an existing translation instead of some null entry.
- If you change or expand your database to encompass other translatable items/things/etc you can use the same tables and system - this is very uncoupled from the rest of the data.

-Adam

Share Edit Follow

edited Nov 25 '08 at 14:36

answered Nov 25 '08 at 9:32




[Adam Davis](#)

88.3k 56 258 328

- 1 I agree, though personally I'd have a localised table for each main table, to allow foreign keys to be implemented. – [Neil Barnwell](#) Dec 3 '08 at 11:58
- 1 Although the third option is the most clean and sound implementation of the problem it is more complex then first one. I think displaying, editing, reporting the general version needs so much extra effort that it does not always acceptable. I have implemented both solutions, the simpler was enough when the users needed a read-only (sometimes missing) translation of the "main" application language. – [rics](#) Oct 14 '09 at 7:40

- 12 What if the product table contains several translated fields ? When retrieving products, you will have to do one additional join per translated field, which will result in severe performance issues. There is as well (IMO) additional complexity for insert/update/delete. The single advantage of this is the lower number of tables. I would go for the method proposed by SunWuKung : I think it's a good balance between performance, complexity, and maintenance issues. – [Frosty Z](#) Jan 20 '11 at 10:08

@rics- I agree, well what do you suggest to ... ? – [saber](#) Sep 8 '11 at 4:50

@Adam- I am confused, maybe I misunderstood. You suggested the third one, right? Please explain it in more detail how are relations between those tables gonna be ? You mean we have to implement Translation and TranslationEntry tables for each tables in DB ? – [saber](#) Sep 8 '11 at 4:57 



Take a look for this example:

9



```
PRODUCTS (
  id
  price
  created_at
)

LANGUAGES (
  id
  title
)

TRANSLATIONS (
  id          (// id of translation, UNIQUE)
  language_id (// id of desired language)
  table_name  (// any table, in this case PRODUCTS)
  item_id     (// id of item in PRODUCTS)
  field_name  (// fields to be translated)
  translation (// translation text goes here)
)
```

I think there's no need to explain, the structure describes itself.

Share Edit Follow

answered Aug 12 '13 at 7:22




[bamburik](#)

107 1 3

this is good. but how would you search (for example product_name) ? – [Illuminati](#) Dec 9 '15 at 23:21

Did you have a live example somewhere of your sample ? Did you get any problems by using it ? – [David Létourneau](#) Jan 11 '16 at 14:11

Sure, I have multilingual real estate project, we support 4 languages. The search is a bit complicated, but its fast. Of course in large projects it might be slower than it needs to be. In small or medium projects its ok. – [bamburik](#) Jan 14 '16 at 10:02 

I usually would go for this approach (not actual sql), this corresponds with your last option.

8

```
table Product
productid INT PK, price DECIMAL, translationid INT FK
```

```
table Translation
translationid INT PK
```

```
table TranslationItem
translationitemid INT PK, translationid INT FK, text VARCHAR, languagecode CHAR(2)
```

```
view ProductView
select * from Product
inner join Translation
inner join TranslationItem
where languagecode='en'
```

Because having all translatable texts in one place makes maintenance so much easier. Sometimes translations are outsourced to translation bureaus, this way you can send them just one big export file, and import it back just as easily.

Share Edit Follow

answered Nov 25 '08 at 9:37



[user39603](#)

2,165 1 15 13

2 What purpose does the Translation table or the TranslationItem.translationitemid column serve? – [DanMan](#) Dec 14 '14 at 11:41

Before going to technical details and solutions, you should stop for a minute and ask a few questions about the requirements. The answers can have a huge impact on the technical solution. Examples of such questions would be:

4

- Will all languages be used all the time?
- Who and when will fill the columns with the different language versions?
- What happens when a user will need a certain language of a text and there is none in the system?



- Only the texts are to be localized or there are also other items (for example PRICE can be stored in \$ and € because they might be different)

Share Edit Follow

answered Nov 25 '08 at 9:59



Aleris

7,691

3

32

42

I know that localization is a much broader topic and I am aware of the issues that you bring to my attention, but currently I am looking for an answer for a very specific problem of schema design. I assume that new languages will be added incrementally and each will be translated almost completely.

– qbeuek Nov 25 '08 at 11:36



I was looking for some tips for localization and found this topic. I was wondering why this is used:

3

```
CREATE TABLE T_TRANSLATION (  
  TRANSLATION_ID  
)
```



So you get something like user39603 suggests:

```
table Product  
productid INT PK, price DECIMAL, translationid INT FK
```

```
table Translation  
translationid INT PK
```

```
table TranslationItem  
translationitemid INT PK, translationid INT FK, text VARCHAR, languagecode CHAR(2)
```

```
view ProductView  
select * from Product  
inner join Translation  
inner join TranslationItem  
where languagecode='en'
```

Can't you just leave the table Translation out so you get this:


```
table Product
productid INT PK, price DECIMAL

table ProductItem
productitemid INT PK, productid INT FK, text VARCHAR, languagecode CHAR(2)

view ProductView
select * from Product
inner join ProductItem
where languagecode='en'
```

[Share](#) [Edit](#) [Follow](#)

answered Aug 6 '12 at 18:00

[randomizer](#)**1,614** 3 14 31

1 Sure. I'd call the ProductItem table something like ProductTexts or ProductL10n though. Makes more sense. – [DanMan](#) Dec 14 '14 at 11:44



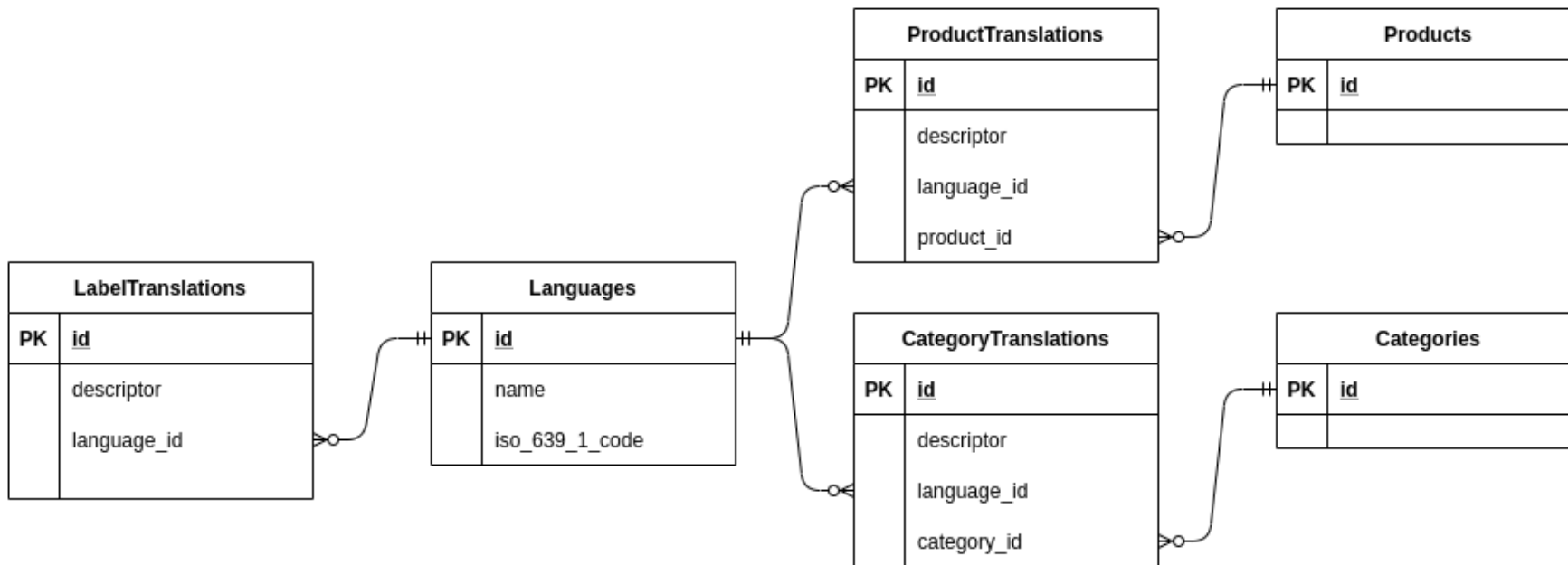
1



You need to remember that when you create a multilanguage database you are cutting off the fields like name or description from the product table and move it to translated resource.

The translated resource may be another table, like here in my example which is designed to work with SQL views for query simplicity and friendly development of the underlying app





Materialized Views / SQL Views

VLabels		
iso_639_1_code	descriptor	value
en	name	name
en	description	description
en	invoice_header_no	Invoice number
en	invoice_header_title	Your invoice
pl	name	nazwa
pl	description	opis
pl	invoice_header_no	Numer faktury
pl	invoice_header_title	Twoja faktura

VCategories			
id	iso_639_1_code	descriptor	value
120	en	name	Slash weapons
120	en	description	Cuts through skin
120	pl	name	Ostrza
120	pl	description	Przecina skórę

I separated LabelTranslations since this is a table with global translations for field labels on the web page. You may call it however you want, they are stateless and don't depend on the specific product or category.

The CategoryTranslations of ProductTranslations are the one that are stateful, this mean that the descriptor of "name" will be the actual product name.

Use materialized views for better performance over simple SQL views (with the cost of storage space and more effort into underlying app development to refresh them), or just go with more heavy SQL Views if you want to.

To create categories materialized view in **Postgres**:

```
CREATE MATERIALIZED VIEW VCategories AS (  
    SELECT cat.id, lng.iso_639_1_code, ct.descriptor, ct.value  
    FROM Categories cat  
    JOIN CategoryTranslations ct ON ct.category_id = cat.id  
    JOIN Languages lng ON lng.id = ct.language_id  
);
```

To query every translation of a category with ID 120

```
SELECT * FROM VCategories WHERE id = 120 AND iso_639_1_code = 'en'
```

I feel it convenient while working with the code of the application, you are able to write a very simple code to query the translations and search for the records

Share Edit Follow

answered Sep 7 '20 at 21:38



Bartłomiej Sobieszek

2,449 2 19 35



1



Would the below approach be viable? Say you have tables where more than 1 column needs translating. So for product you could have both product name & product description that need translating. Could you do the following:

```
CREATE TABLE translation_entry (  
    translation_id    int,  
    language_id      int,
```



```
table_name      nvarchar(200),
table_column_name nvarchar(200),
table_row_id     bigint,
translated_text  ntext
)

CREATE TABLE translation_language (
  id int,
  language_code CHAR(2)
)
```

Share Edit Follow

answered Dec 13 '12 at 16:03

[davey](#)**1,502** 4 25 41

I agree with randomizer. I don't see why you need a table "translation".

1

I think, this is enough:



```
TA_product: ProductID, ProductPrice
TA_Language: LanguageID, Language
TA_Productname: ProductnameID, ProductID, LanguageID, ProductName
```

Share Edit Follow

edited Oct 4 '12 at 9:31

[Sergey K.](#)**23.6k** 13 95 168

answered Aug 21 '12 at 20:35

[Bart VW](#)**11** 1

This [document](#) describes the possible solutions and the advantages and disadvantages of each method. I prefer the "row localization" because you do not have to modify the DB schema when adding a new language.

1

Share Edit Follow



answered Mar 23 '20 at 3:50

[Jaska](#)**922** 8 9



0

"Which one is best" is based on the project situation. The first one is easy to select and maintain, and also the performance is best since it don't need to join tables when select entity. If you confirmed that your poject is just only support 2 or 3 languages, and it will not increase, you can use it.



The second one is okey but is hard to understand and maintain. And the performance is worse than first one.



The last one is good at scalability but bad at performance. The T_TRANSLATION_ENTRY table will become larger and larger, it's terrible when you want to retrieve a list of entities from some tables.

Share Edit Follow

answered Apr 3 '13 at 5:34



studyzy

13 3



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.