When and How You Should Denormalize a Relational Database

- **175797 VIEWS**
- **13 MIN**
- **JAN 13, 2020**


Gleb B.
Copywriter

Alex B.
Ruby/JS Developer

Website speed is a priority for businesses in 2020.

Faster websites rank higher on search engines and also provide better user experiences, resulting in higher conversion rates. No wonder website owners demand faster page loading speeds – leaving developers to make it happen.

Database optimization is an essential step to improve website performance. Typically, developers normalize a relational database,

meaning they restructure it to reduce data redundancy and enhance data integrity. However, sometimes normalizing a database isn’t enough, so to improve database performance even further developers go the other way around and resort to database denormalization.

In this article, we take a closer look at denormalization to find out when this method is appropriate and how you can do it.

When to denormalize a database

What is database denormalization? Before diving into the subject, let’s emphasize that normalization still remains the starting point, meaning that you should first of all normalize a database’s structure. The essence of normalization is to put each piece of data in its appropriate place; this ensures data integrity and facilitates updating. However, retrieving data from a normalized database can be slower, as queries need to address many different tables where different pieces of data are stored. Updating, to the contrary, gets faster as all pieces of data are stored in a single place.

The majority of modern applications need to be able to retrieve data in the shortest time possible. And that’s when you can consider denormalizing a relational database. As the name suggests, denormalization is the opposite of normalization. When you normalize a database, you organize data to ensure integrity and eliminate redundancies. Database denormalization means you deliberately put the same data in several places, thus increasing redundancy.

“Why denormalize a database at all?” you may ask. The main purpose of denormalization is to significantly speed up data retrieval. However, denormalization isn’t a magic pill. Developers should use this tool only for particular purposes:

1 To enhance query performance

Typically, a normalized database requires joining a lot of tables to fetch queries; but the more joins, the slower the query. As a countermeasure, you can add redundancy to a database by copying values between parent and child tables and, therefore, reducing the number of joins required for a query.

#2 To make a database more convenient to manage

A normalized database doesn’t have calculated values that are essential for applications. Calculating these values on-the-fly would require time, slowing down query execution.

You can denormalize a database to provide calculated values. Once they’re generated and added to tables, downstream programmers can easily create their own reports and queries without having in-depth knowledge of the app’s code or API.

#3 To facilitate and accelerate reporting

Often, applications need to provide a lot of analytical and statistical information. Generating reports from live data is time-consuming and can negatively impact overall system performance.

Denormalizing your database can help you meet this challenge. Suppose you need to provide a total sales summary for one or many users; a normalized database would aggregate and calculate all invoice details multiple times. Needless to say, this would be quite time-consuming, so to speed up this process, you could maintain the year-to-date sales summary in a table storing user details.

Database denormalization techniques

Now that you know when you should go for database denormalization, you're probably wondering how to do it right. There are several denormalization techniques, each appropriate for a particular situation. Let's explore them in depth:

Storing derivable data

If you need to execute a calculation repeatedly during queries, it's best to store the results of it. If the calculation contains detail records, you

should store the derived calculation in the master table. Whenever you decide to store derivable values, make sure that denormalized values are always recalculated by the system.

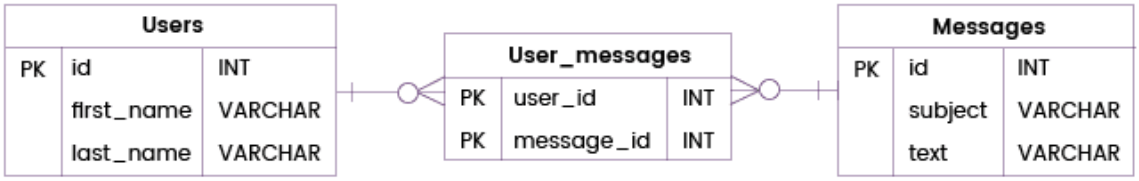
Here are situations when storing derivable values is appropriate:

- When you frequently need derivable values
- When you don’t alter source values frequently

Advantages	Disadvantages
No need to look up source values each time a derivable value is needed	Running data manipulation language (DML) statements against the source data requires recalculation of the derivable data
No need to perform a calculation for every query or report	Data inconsistencies are possible due to data duplication

Example

Normalized database



Denormalized database

As an example of this denormalization technique, let's suppose we're building an email messaging service. Having received a message, a user gets only a pointer to this message; the pointer is stored in the **User_messages** table. This is done to prevent the messaging system from storing multiple copies of an email message in case it's sent to many different recipients at a time. But what if a user deletes a message from their account? In this case, only the respective entry in the **User_messages** table is actually removed. So to completely delete the message, all **User_messages** records for it must be removed.

Denormalization of data in one of the tables can make this much simpler: we can add a **users_received_count** to the **Messages** table

to keep a record of **User_messages** kept for a specific message. When a user deletes this message (read: removes the pointer to the actual message), the **users_received_count** column is decremented by one. Naturally, when the **users_received_count** equals zero, the actual message can be deleted completely.

Using pre-joined tables

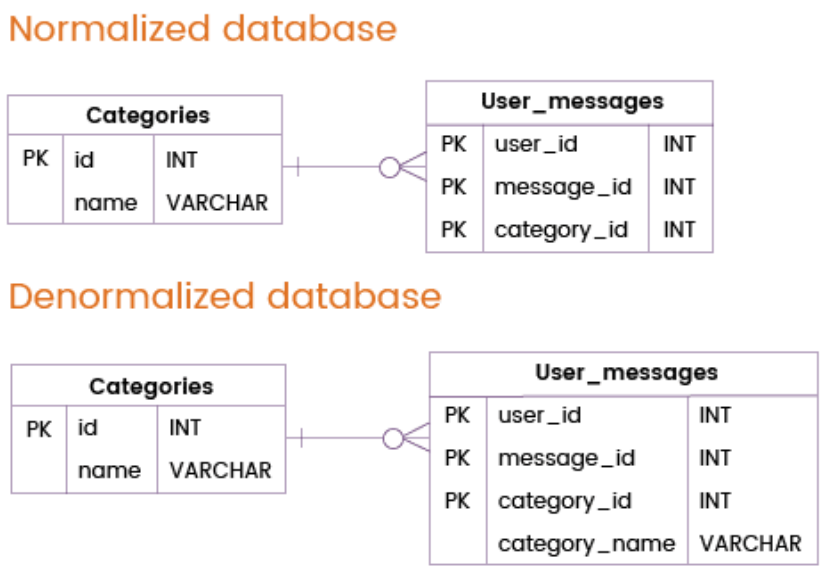
To pre-join tables, you need to add a non-key column to a table that bears no business value. This way, you can dodge joining tables and therefore speed up queries. Yet you must ensure that the denormalized column gets updated every time the master column value is altered.

This denormalization technique can be used when you have to make lots of queries against many different tables – and as long as stale data is acceptable.

Advantages	Disadvantages

No need to use multiple joins	DML is required to update the non-denormalized column
You can put off updates as long as stale data is tolerable	An extra column requires additional working and disk space

Example



Imagine that users of our email messaging service want to access messages by category. Keeping the name of a category right in the **User_messages** table can save time and reduce the number of necessary joins.

In the denormalized table above, we introduced the **category_name** column to store information about which category each record in the **User_messages** table is related to. Thanks to denormalization, only a

query on the **User_messages** table is required to enable a user to select all messages belonging to a specific category. Of course, this denormalization technique has a downside – this extra column may require a lot of storage space.

Using hardcoded values

If there's a reference table with constant records, you can hardcode them into your application. This way, you don't need to join tables to fetch the reference values.

However, when using hardcoded values, you should create a check constraint to validate values against reference values. This constraint must be rewritten each time a new value in the A table is required.

This data denormalization technique should be used if values are static throughout the lifecycle of your system and as long as the number of these values is quite small. Now let's have a look at the pros and cons of this technique:

Advantages	Disadvantages

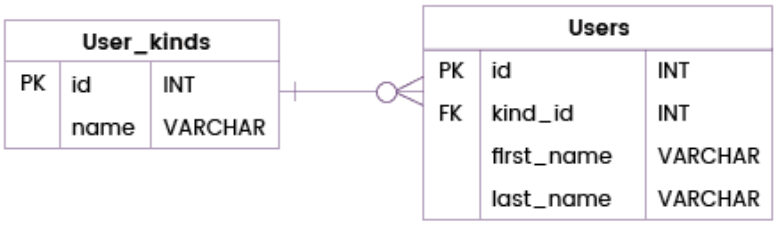
No need to implement a lookup table

No joins to a lookup table

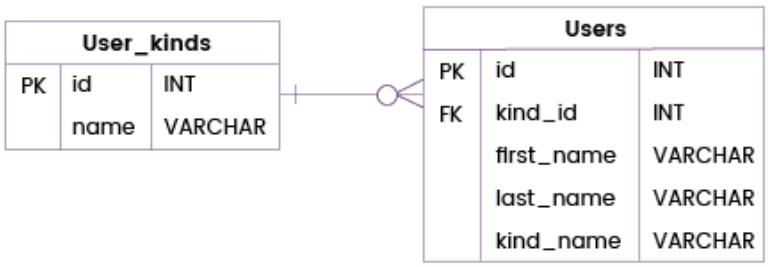
Recoding and restating are required if look-up values are altered

Example

Normalized database



Denormalized database



Suppose we need to find out background information about users of an email messaging service, for example the kind, or type, of user. We've created a **User_kinds** table to store data on the kinds of users we need to recognize.

The values stored in this table aren't likely to be changed frequently, so we can apply hardcoding. We can add a check constraint to the

column or build the check constraint into the field validation for the application where users sign in to our email messaging service.

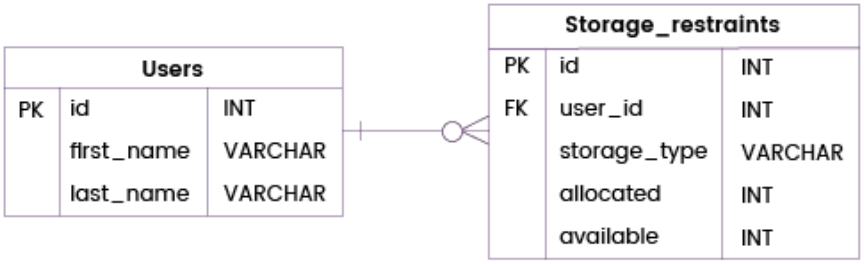
Keeping details with the master

There can be cases when the number of detail records per master is fixed or when detail records are queried with the master. In these cases, you can denormalize a database by adding detail columns to the master table. This technique proves most useful when there are few records in the detail table.

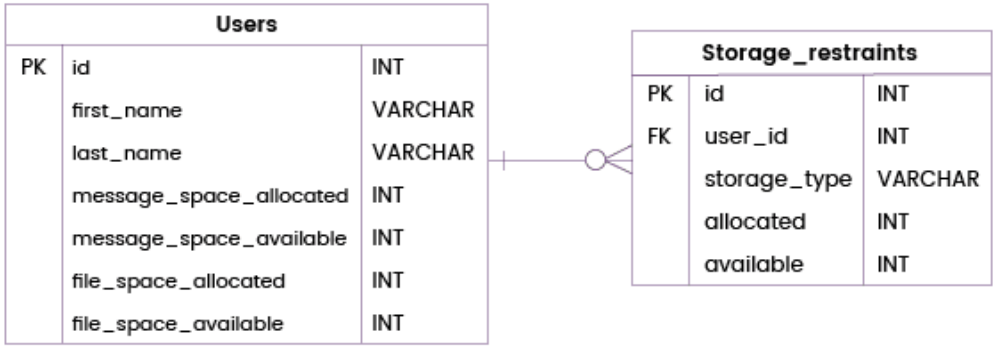
Advantages	Disadvantages
No need to use joins	Increased complexity of DML
Saves space	

Example

Normalized database



Denormalized database



Imagine that we need to limit the maximum amount of storage space a user can get. To do so, we need to implement restraints in our email messaging service – one for messages and another for files. Since the amount of allowed storage space for each of these restraints is different, we need to track each restraint individually. In a normalized relational database, we could simply introduce two different tables – **Storage_types** and **Storage_restraints** – that would store records for each user.

Instead, we can go a different way and add denormalized columns to the **Users** table:

message_space_allocated

message_space_available

file_space_allocated

file_space_available

In this case, the denormalized **Users** table stores not only the actual information about a user but the restraints as well, so in terms of functionality the table doesn’t fully correspond to its name.

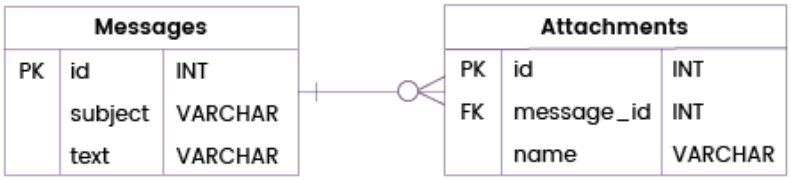
Repeating a single detail with its master

When you deal with historical data, many queries need a specific single record and rarely require other details. With this database denormalization technique, you can introduce a new foreign key column for storing this record with its master. When using this type of denormalization, don’t forget to add code that will update the denormalized column when a new record is added.

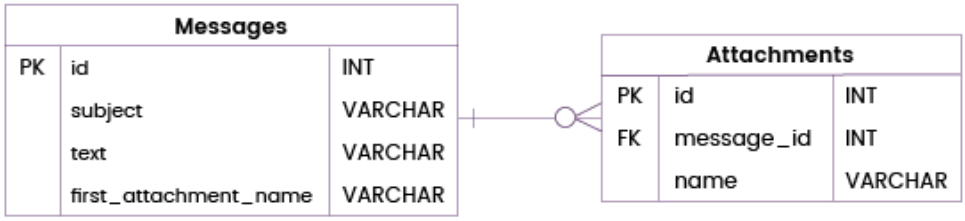
Advantages	Disadvantages
No need to create joins for queries that need a single record	Data inconsistencies are possible as a record value must be repeated

Example

Normalized database



Denormalized database



Often, users send not only messages but attachments too. The majority of messages are sent either without an attachment or with a single attachment, but in some cases users attach several files to a message.

We can avoid a table join by denormalizing the **Messages** table through adding the **first_attachment_name** column. Naturally, if a message contains more than one attachment, only the first

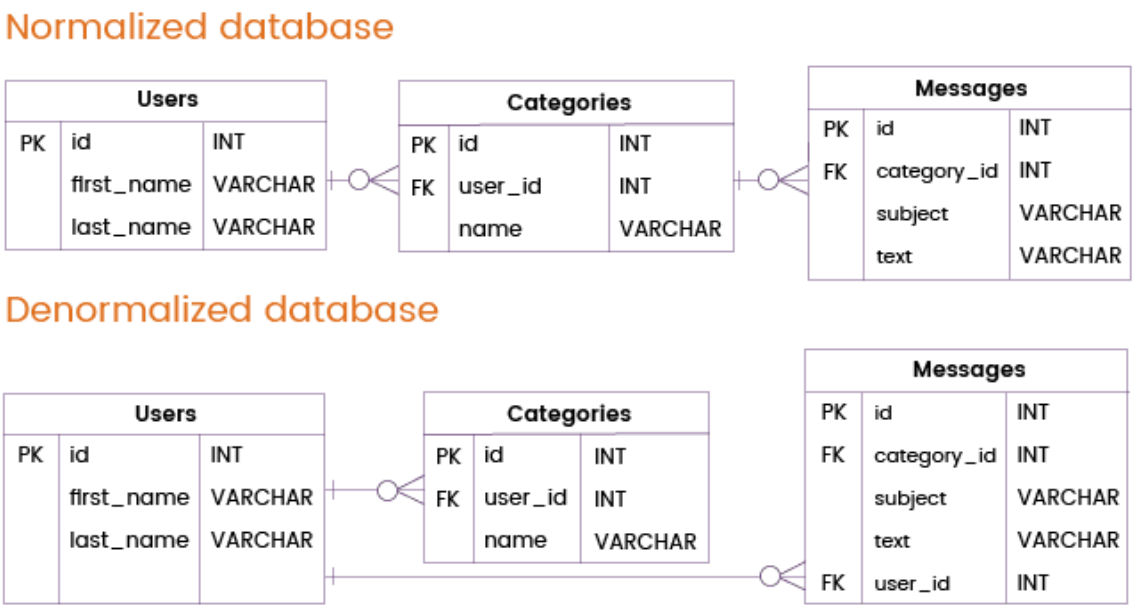
attachment will be taken from the **Messages** table while other attachments will be stored in a separate **Attachments** table and, therefore, will require table joins. In most cases, however, this denormalization technique will be really helpful.

Adding short-circuit keys

If a database has over three levels of master detail and you need to query only records from the lowest and highest levels, you can denormalize your database by creating short-circuit keys that connect the lowest-level grandchild records to higher-level grandparent records. This technique helps you reduce the number of table joins when queries are executed.

Advantages	Disadvantages
Fewer tables are joined during queries	Need to use more foreign keys
	Need extra code to ensure consistency of values

Example



Now let’s imagine that an email messaging service has to handle frequent queries that require data from the **Users** and **Messages** tables only, without addressing the **Categories** table. In a normalized database, such queries would need to join the **Users** and **Categories** tables.

To improve database performance and avoid such joins, we can add a primary or unique key from the **Users** table directly to the **Messages** table. This way we can provide information about users and messages without querying the **Categories** table, which means we can do without a redundant table join.

Drawbacks of database denormalization

Now you’re probably wondering: to denormalize or not to denormalize?

Though denormalization seems like the best way to increase performance of a database and, consequently, an application in general, you should resort to it only when other methods prove inefficient. For instance, often insufficient database performance can be caused by incorrectly written queries, faulty application code, inconsistent index design, or even improper hardware configuration.

Denormalization sounds tempting and extremely efficient in theory, but it comes with a number of drawbacks that you must be aware of before going with this strategy:

- **Extra storage space**

  When you denormalize a database, you have to duplicate a lot of data. Naturally, your database will require more storage space.
- **Additional documentation**

  Every single step you take during denormalization must be

properly documented. If you change the design of your database sometime later, you'll need to revise all rules you created before: you may not need some of them or you may need to upgrade particular denormalization rules.

- **Potential data anomalies**

  When denormalizing a database, you should understand that you get more data that can be modified. Accordingly, you need to take care of every single case of duplicate data. You should use triggers, stored procedures, and transactions to avoid data anomalies.

- **More code**

  When denormalizing a database you modify select queries, and though this brings a lot of benefits it has its price – you need to write extra code. You also need to update values in new attributes that you add to existing records, which means even more code is required.

- **Slower operations**

  Database denormalization may speed up data retrievals but at the same time it slows down updates. If your application needs to perform a lot of write operations to the database, it may show slower performance than a similar normalized database. So make

sure to implement denormalization without damaging the usability of your application.

Database denormalization tips

As you can see, denormalization is a serious process that requires a lot of effort and skill. If you want to denormalize databases without any issues, follow these useful tips:

1. Instead of trying to denormalize the whole database right away, focus on particular parts that you want to speed up.
2. Do your best to learn the logical design of your application really well to understand what parts of your system are likely to be affected by denormalization.
3. Analyze how often data is changed in your application; if data changes too often, maintaining the integrity of your database after denormalization could become a real problem.
4. Take a close look at what parts of your application are having performance issues; often, you can speed up your application by fine-tuning queries rather than denormalizing the database.
5. Learn more about data storage techniques; picking the most relevant can help you do without denormalization.

Final thoughts

You should always start from building a clean and high-performance normalized database. Only if you need your database to perform better at particular tasks (such as reporting) should you opt for denormalization. If you do denormalize, be careful and make sure to document all changes you make to the database.

Before going for denormalization, ask yourself the following questions:

- Can my system achieve sufficient performance without denormalization?
- Might the performance of my database become unacceptable after I denormalize it?
- Will my system become less reliable?

If your answer to any of these question is yes, then you'd better do without denormalization as it's likely to prove inefficient for your application. If, however, denormalization is your only option, you should first normalize the database correctly, then move on to

denormalizing it, carefully and strictly following the techniques we've described in this article.

For more insights into the latest trends in software development, subscribe to our blog.