High-available systems make heavy use of redundancy to avoid downtime if one instance crashes. The redundancy of the application plays a critical role during the migration. It enables you to perform a rolling update.

The implementation of a rolling update depends on your technology stack. But the idea is always the same: You have a bunch of instances of a subsystem and you shutdown, update, and restart one instance after the other. While doing that, you run the old and the new version of your application in parallel.

Multi-Step Migration Process

The rolling update adds a few requirements to your database migration. You no longer need to just adapt the database in the way it's required by your application; you also need to do it in a way that the old and the new version of your application can work with the database. That means that all migrations need to be backward-compatible as long as you're running at least one instance of the old version of your application. But not all operations, e.g., renaming or removing a column, are backward compatible. These operations require a multi-step process that enables you to perform the migration without breaking your system.

Backward-Compatible Operations

Backward-compatible operations are all operations that change your database in a way that it can be used by the old and the new version of your application.

Add a table or a view

Adding new tables or views doesn't affect the old instances of your application. Just keep in mind that while you're performing the rolling update, some users might trigger write operations on old application instances. You might need to clean up your data and add missing records to the new table after all application instances have been migrated.

Add a column

Adding a new column without a not null constraint is also a backward compatible operation that you can perform without any risk.

If you need to introduce a new column with a not null constraint, you either need to provide a default value for all old records or follow this process:

1. Add the column without a default value and update all application instances.
2. Run a database script to fill that field in all existing records.
3. Add the not null constraint.

Remove a column that's not used by the old and the new version of your application

Removing a database column that is neither accessed by the old nor the new version of your application is also a backward-compatible operation.

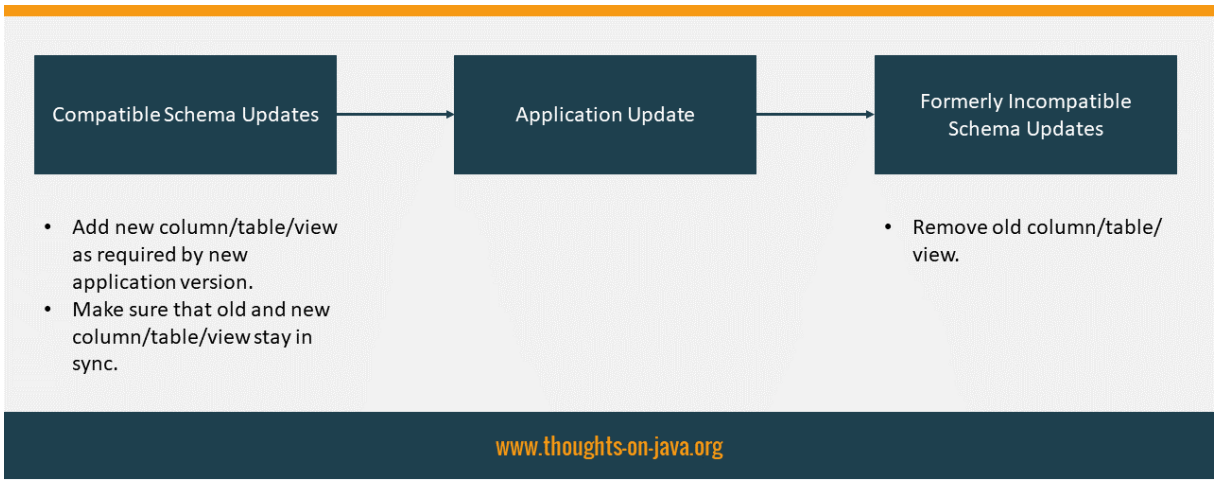
Remove constraints

The removal of the constraint itself is a backward-compatible operation. The old version of your application can still write to the database in the same way as it did before.

But you need to check if there are any old use case implementations that would break if any database record doesn't fulfill the constraint. I don't know any good way to solve this issue. Your only option is to remove the constraint and to roll-out the update quickly.

Backward-Incompatible Operations

Backward-incompatible operations are all the operations that change your database schema in a way that it can no longer be used by the old version of your application. You need to break these operations into a backward-compatible part which you perform before you update your application and a second part that you execute after you updated all application instances. In most cases, that requires you to add a new column or table in the first and to remove the old one in a later step.



Rename a column, a table or a view

The main issues of this migration occur during the rolling update of your application. While doing that, you're running old and new versions of your application in parallel. The old version is still using the old database column, and the new one is using the new column. So, you need to make sure that both use the same data and that you don't lose any write operations.

Option 1: Sync with database triggers

1. Add a column with the new name and the same data type as the old one. You then copy all data from the old column to the new one.
   You also need to add database triggers to keep both columns in sync so that neither the old nor the new version of your application works on outdated data.
2. Perform a rolling update of all application instances.
3. Remove the old database column and the database triggers.

Option 2: Sync programmatically

1. Add a column with the new name and the same data type as the old one. You then copy all data from the old column to the new one.
2. Make sure that the new version of your application reads from and writes to the old and the new database column. Let's call this version *new1*.

Please also keep in mind that there are still old instances of your application that don't know about the new column and that can write new and update existing records at any time. As your database doesn't synchronize the write operations, you need to do that in the code of version *new1*.
After you've made sure that the *new1* version of your application can handle this situation, you can perform a rolling update of all application instances.

3. All of your application instances now run version *new1* which knows about the new database column. You can now perform a rolling update to application version *new2* which only uses the new database column.
4. Remove the old database column.

Change the data type of a column

You can change the data type of a column in almost the same way as you rename the column.

Remove a column or table or view that's still used by the old version of your application

I'm sorry to tell you that you can't remove that column/table/view. At least not now. You first need to update your application so that there is no running instance of it that still uses it. After you've done that, you can remove the no longer used column/table/view from your database.