Expand and Contract

A Pattern to Apply Breaking Changes to Persistent Data with Zero Downtime

Tim Wellhausen
kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

v1.0, July 18, 2018

Intent

The Expand and Contract pattern provides a way to implement breaking changes to a system in a safe manner. The pattern is particularly helpful in an environment where maintenance downtime is unacceptable. This paper examines the details on how to apply the pattern when the structure of persistent data needs to be changed. The paper is supposed to be a useful read for every software developer who operates in such an environment.

Introduction

The Expand and Contract pattern, also known as Parallel Change, has already been documented elsewhere ([1], [2]). The pattern provides an approach to changing interfaces without breaking systems. The pattern is applicable to all kinds of interfaces, such as interfaces in a programming language, REST APIs or database schemas.

The pattern's main idea is to first expand an interface by introducing a new structure without breaking the old one. The old structure is kept in parallel as long as some client code still accesses it. Once the old structure is not needed any more, it can be safely removed.

Applying the pattern on persistent data in a database is particularly challenging because existing data needs to be migrated from the old into the new structure. Doing this was still fairly straight-forward at the times when you could shut down a system temporarily to do some maintenance. Nowadays, many systems need to operate 24/7 without any maintenance downtime and structural changes of persistent data may take a long time due to the size of the data to migrate.

This paper therefore focuses on the applicability of the pattern in the context of changing the structure of persistent data. While the pattern probably can be applied to various types of databases, this paper assumes that the data to change resides in a relational or document database. This means that there is an explicit or implicit data schema that the application that operates on the data can rely on.

In order to ensure that a system is running all the time while transitioning data from an old into a new structure, the overall changes need to be deployed in several distinct steps. As these steps are crucial to succeed in applying the pattern, they are examined in detail. An example gives further information.

Let's start by rephrasing the pattern and putting it into the required context.

The Expand and Contract Pattern

A system operates on a central database. The database, which could be a relational or a document database, contains data in some structured form based on an explicit or implicit schema. Over time, system requirements change and by implementing those changes, the database schema and thus the actual data needs to be modified. Some schema modifications are backwards-compatible in nature (such as adding new tables/collections or columns/fields). Other modifications break existing code (such as changing the cardinality between some entities from 1:n to n:m).

In order to achieve 24/7 operations, all system components exist redundantly. In particular, there are multiple instances of the database system and multiple instances of the application server. Deployments of updated system components are done instance by instance, which means that some instances are unavailable during a system update. The system might be a large monolithic application, a small microservice or anything in between.

Implementing breaking changes to the structure of persistent data must not interfere with the operation of the system as a whole.

The following *forces* make it difficult to find a good solution.

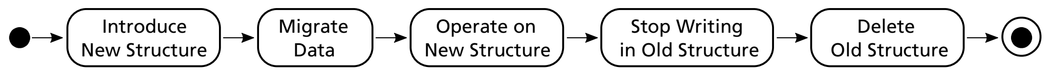
- *Uptime*. The system must be in operation all the time. Maintenance downtime to perform isolated data migrations and code deployments are not available.
- *Time-to-market*. Code changes need to be deployed as soon as possible. Continuous deployment is reality, frequent deployments are the norm.
- *Testing*. Things can go wrong. You therefore want to reduce the risk of breaking the system by deploying code changes to some nodes first in order to test them. Rolling forward is the typical attitude to fix problems.
- *Rollback*. Still, sometimes things can go so wrong that rolling back the latest change is the only option to quickly restore the system to a healthy state. Even when you implement breaking changes to the database, it should be possible to roll back.
- *Large data sets*. Migrating data from an old structure into a new structure takes some time. The larger the data set is, the longer migrations run. Still, a migration must not interrupt the operation of the production system.
- *Computational resources.* A successful application often is in heavy use. Scaling up the application and database instances is not always an option. Thus, additional load and space usage should be avoided.
- *Consistent data*. During the deployment of a new release, typically there is a phase in which some application instances still run old code whereas other application instances already run new code. Data records must not break even if different versions of the code run against the database.
- *Consistent data structure*. All data in the database should eventually be kept in the same structure. Even if the database allows entities in the same space to have a different structure (e.g. in a collection of documents), this is not desirable as it would complicate the code to cover all those different structures in the long term.

* * *

Therefore:

Implement breaking changes in multiple steps so that each individual step does not break the system and can be reverted. First, expand the system by adding the new structure to the database. Then migrate the existing data into the new structure while the system redundantly writes into both the old and the new structure. After the migration is done, contract the system to remove the old data structure and the old code.

The following diagram outlines these steps that are described in detail below:



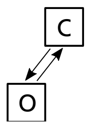
Each step needs to deployed separately and after each step, you need to make sure that the system is fully operational before you can apply the next step.

Every step is illustrated by a small diagram that shows the state after the step has been applied.

Step 0: Initial state

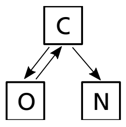
Initially, the application code (*C*) performs some read and write operations on a data structure (the old structure – *O*).

The new structure (*N*) has been thought through but does not exist yet in the database.



Step 1: Introduce new structure and update application to write into both structures

The first step introduces the new data structure (i.e. new tables/collections, new columns/fields) to the database. In case of a relational database, the new structure needs to be explicitly created. The application code needs to be changed in such a way that data is written into both the old and the new structure whereas the system continues to read the data from the old structure.



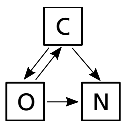
This means, from now on the database stores some data redundantly until the old structure is deleted by the last step, thereby increasing the space that the data consumes. This also means that the number of write operations on the database increases during this period.

The new structure can be prepared to hold data that did not exist before. However, some constraints cannot be enforced yet, e.g. mandatory columns or fields, as the system still operates on the old structure, i.e. there is not yet any data available to store into new fields (unless it can be derived from existing data).

At this point, the old data structure is written as before. All new and changed data records also get written into the new structure. This step can be easily rolled back. If necessary, the new data structure can be removed from the database.

Step 2: Migrate data into the new structure

Now, all relevant data needs to be migrated from the old into the new structure. The old structure is still kept untouched. Depending on the data size, this step may take a considerable time. It thus needs to run in the background, maybe deployed on a separate server instance. This process does not interfere with the correctness of the application's operations because the application still only reads data from the old structure. However, the migration could have an impact on the performance of the production system.

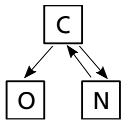


The migration process may skip those entities that have already been written into the new structure as part of some data modifications after the deployment of step 1. In order to ensure data consistency, the migration code must produce the same result as the application code deployed by step 1. However, some error handling (such as optimistic or pessimistic locking) is important to cover the case when both the application and the migration touch the same data records.

This step can also be easily reverted. If necessary, the new data structure needs to be cleaned and the migration started anew.

Step 3: Operate on new structure

This step makes the transition to read data from the new structure while still writing into both the old and the new structure.



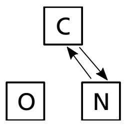
The first two steps ensured that data is now consistently available both in the old and the new structure. Thus, reading the data from either source leads to the same result. Rolling back this step therefore can also be safely done.

The reason to change data typically comes from some changed business requirements that also introduce new features to a system. If this step involves the rollout of a new UI, then reverting this step is noticeable to the users.

If some new fields were already introduced to implement the new functionality in step 1, then it is possible to fill these fields now and enforce some missing constraints. Be aware that in such a case, it is not possible any more to keep the old data structure completely consistent with the new structure (because the old structure does not have these new fields). Rolling back from this step could result in loosing data that was generated by the new functionality. As alternative, you could postpone the introduction of new fields until this step has been successfully deployed.

Step 4: Stop writing into old structure

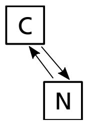
The first step to contract the system is to cease from writing into the old structure. From now on, database reads and writes both affect only the new structure.



Reverting this step cannot be easily done anymore other by restoring data from a backup. So, reverting also comes with a data-loss. However, there should be little need to revert as the old structure was not in use any more after the preceding step.

Step 5: Delete old structure

Once the preceding step has been rolled out to all servers, then there is no write operation on the old structure any more. Thus, now, the old structure can be safely deleted from the database.



* * *

Following the Expand and Contract pattern brings a couple of *advantages*.

- *Uptime*. Incompatible changes to the data structure can be deployed without any downtime of the system.
- *Testing*. The pattern requires the ability to run application instances with old code and new code in parallel. This makes it easy to deploy code changes to a few instances first in order to test it.
- *Rollback*. Every single step can be rolled back once it has been deployed. However, it is not guaranteed to revert multiple steps.
- *Large data sets*. The data migration is performed as a step separate from all others and can be executed in the background. It does not modify any data that is read by the production system and should therefore not interfere with the production system. Even if the migration process takes a long time, this just extends the time until the next step can be taken.
- *Consistent data*. In the end, all data is consistent and redundancy-free.
- *Consistency data structure*. After applying all steps, all data is stored in the new data structure only.

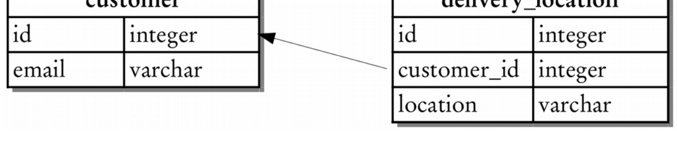
However, the pattern also has *liabilities*.

- *Time-to-market*. If done properly, the deployment of a breaking change needs some time until all steps have been successfully finished. Even small changes that could have been done quickly in the old times where system downtime was available, can now take days or weeks until they are fully rolled out.
- *Computational resources.* While the system is expanded, some data is kept redundantly, therefore increasing the space that the data occupies. At the same time, the number of write operations increases as the application writes into both the old and the new structure. Additionally, the migration process adds load to the database.
- *Consistent data*: In between the transition, data is available redundantly in the old and the new structure. This could be a problem if data is modified by some tools outside the scope of the system, e.g. by some manual intervention.
- *Effort*. Because of the lengthy process to perform each step individually, the application of the pattern involves a lot of effort. There might be a desire to take shortcuts. In some situations, it may be feasible to combine steps (e.g. if writing data is only done once a day by some nightly job). In other cases, it may introduce subtle data inconsistencies that are not obvious when they happen.

Example

To give an example, suppose you are a developer at an online store. The company's competitive advantage has always been its ability to quickly improve the customers' experience whenever there was an opportunity. Needless to say that the online store has to run 24/7 to never loose a customer because of some system maintenance downtime. There is a cluster of application instances and another cluster of database instances.

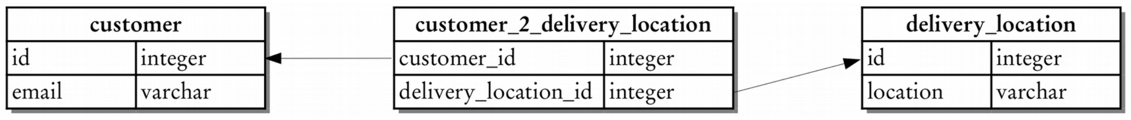
Also suppose that all customer data is stored in a relational database system. Data about a customer is kept in a possibly very large customer table. For every customer, the system keeps track of multiple delivery locations so that the customer can choose among those locations when placing an order. One such location may be the customer's home address, another the place of work. These delivery locations are stored in a separate table with a foreign key reference that refers to the customer table as shown in the following diagram:



Existing data structure with 1:n relationship

One day, your product owner explains to you a new feature that the company would like to implement. In many cities, the company is going to set up delivery locations at places such as grocery stores and gas stations. The idea is to let the customers choose such a location for the delivery of goods so that they can pick up their parcels there whenever it is convenient to them. In the online store, the system should suggest such locations to customers who live nearby one of them. For a customer that chooses such a location, it should behave and feel just like another personal delivery address.

As the data of these delivery locations need to be maintained by the company, it is not feasible to keep the current data structure as that would make it necessary to create a new entry for every customer. The current 1:n relationship between customers and delivery locations therefore needs to be changed into an n:m relationship so that multiple customers can share the same delivery location.



Target data structure with n:m relationship

As you know about the Expand and Contract pattern, you start the development right away. You decide that, at first, you only transform the data structure without introducing any new features.

Step 1. In the database you leave both the customer and the delivery_location tables as they are but you add the new mapping table customer_2_delivery_location. This change can be deployed into production at once. In parallel, you expand the code so that for every new delivery location an entry is written into the mapping table. Look-ups for delivery locations are not touched, i.e. the system still evaluates the delivery_location table's foreign key reference. But from now on, the system creates mapping entries for all new delivery locations.

Step 2. Next thing to do is to migrate all existing delivery locations. The migration is quite straightforward: For every delivery address in the system, you need to write a new entry into the new mapping table. As the production system does the same for new delivery locations, you need to take care not to create duplicate entries. The migration is run on a separate server instance not to interfere with the production system. As the customer table is large, it takes a while until the new mapping table is filled.

Step 3. Once the migration has completed, the new data structure is fully initialized and can be used to read from. The system keeps on writing both the foreign key into the delivery_location table and creating mapping entries in customer_2_delivery_location. However, operations to fetch delivery locations for a customer are changed so that only the mapping data in customer_2_delivery_location is taken into consideration. In order to test the changes in production, the new code is first deployed to a single instance of the application cluster. After verifying that everything works fine, the code is deployed to all other instances one by one.

Step 4. From now on, there are no read operations any more on the foreign key in delivery_location. But the system still writes into it. So, the next deployment removes the foreign key constraint from the table column and disables writing the foreign key. The corresponding field can also be removed from any domain-related code.

Step 5. Once the latest deployment has been rolled out to all servers, there are no write operations any more on the foreign key column in delivery_location. Now it is safe to remove the column alltogether.

Finally, the new mapping table between customers and delivery locations is fully in place and the system is prepared for the implementation of the actual new requirements.

Conclusion

The requirement to keep a system up and running all the time gives software developers a hard time when they need to implement breaking changes to the database. The Expand and Contract pattern describes an approach that allows such changes to be done in a safe manner. This paper described in details how to apply it by performing a number of steps, resulting in up to five deployments until the changes are fully in effect.

Acknowledgments

I'd like to thank Uwe van Heesch for his insightful review comments that helped me a lot to improve the paper. Also, I'd like to thank Julio Moreno, Fei Li, Tiago Boldt Sousa and Andreas Seitz for their valuable contributions at EuroPLoP 2018.

References

[1] Michael T. Nygard: Release It!, Chapter on Zero Downtime Depoyments, O'Reilly, 2007

[2] Danilo Sato: Parallel Change, https://martinfowler.com/bliki/ParallelChange.html