Les Patterns des Géants du Web – Zero Downtime Deployment

Posté le 22/01/2013 par *Mathieu Poignant*

Description

Dans l'article Continuous Deployment, nous avons vu comment améliorer le *Time To Market,* tout en garantissant la qualité des développements.

L'étape suivante est de garantir que ces déploiements fréquents n'impactent pas la disponibilité du site.

Et c'est là qu'intervient le *Zero Downtime Deployment (ZDD),* qui permet de déployer une nouvelle version d'un système sans interrompre le bon fonctionnement du service.

Mais comment s'assurer d'un déploiement « sans accroc » ?

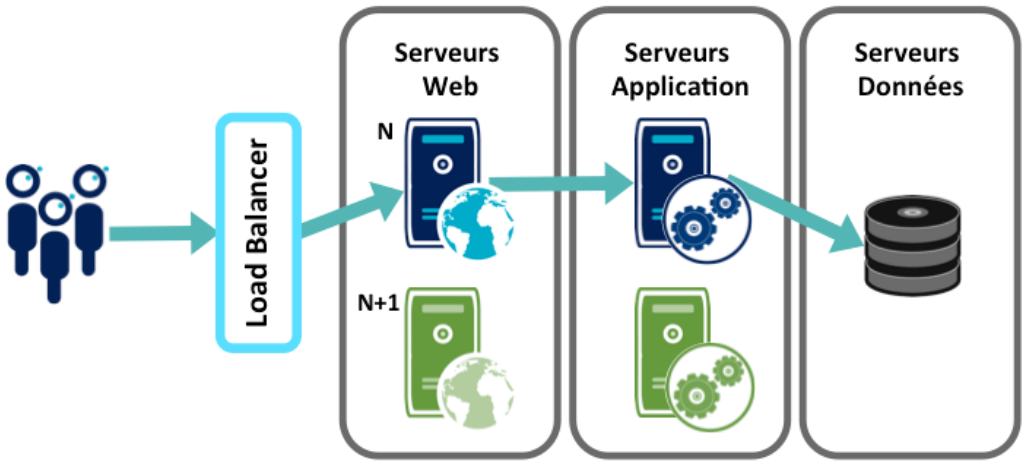
Le ZDD en pratique

La mise en oeuvre du *Zero Downtime Deployment* se base sur un certain nombre de patterns et de bonnes pratiques.

Principaux Patterns

Blue/Green Deployment

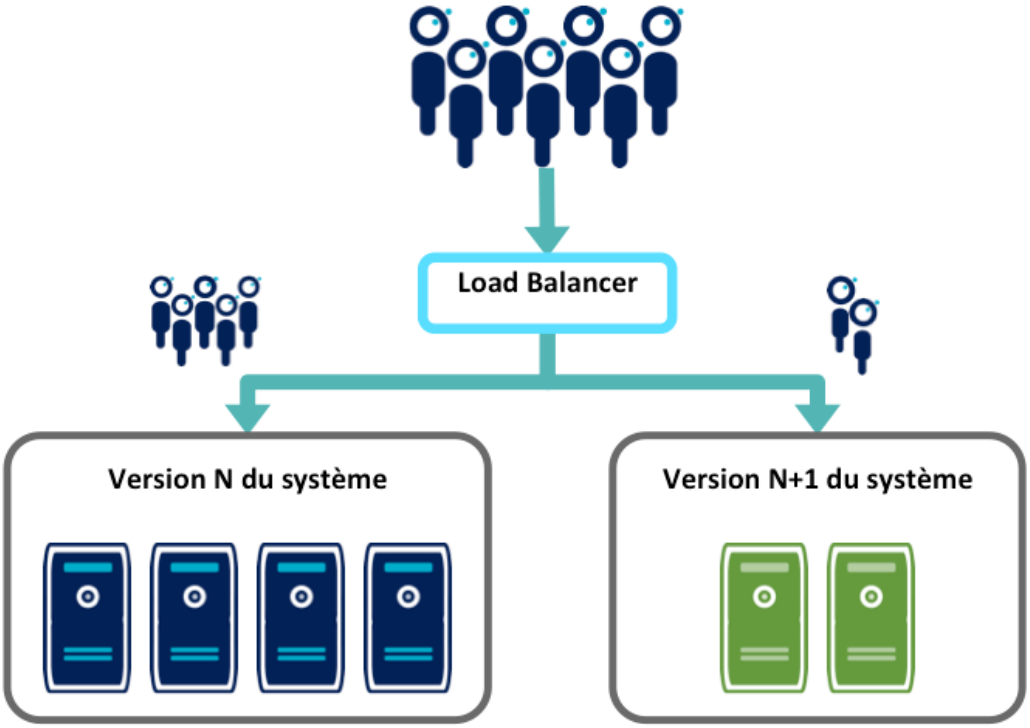
C'est le pattern classique de *ZDD*. Il suppose que l'application soit hébergé sur au moins deux chaînes applicatives, puisque l'objectif est de déployer la version N+1 d'une application sur une des chaînes, tandis que le service est maintenu sur les chaînes encore en version N.



Pattern associé : Canary Release

Ce pattern permet de confronter la version N+1 à une population restreinte d'utilisateurs, tandis que la majorité des utilisateurs ont accès à la version N. Les mécanismes sont identiques au *Blue/Green Deployment*.

Ce pattern est utilisé par Facebook, qui déploie ses mises à jours dans un premier temps à l'ensemble de ses employés, puis à tous les utilisateurs si tout se passe bien pendant une journée.



Pattern associé : Dark Launch

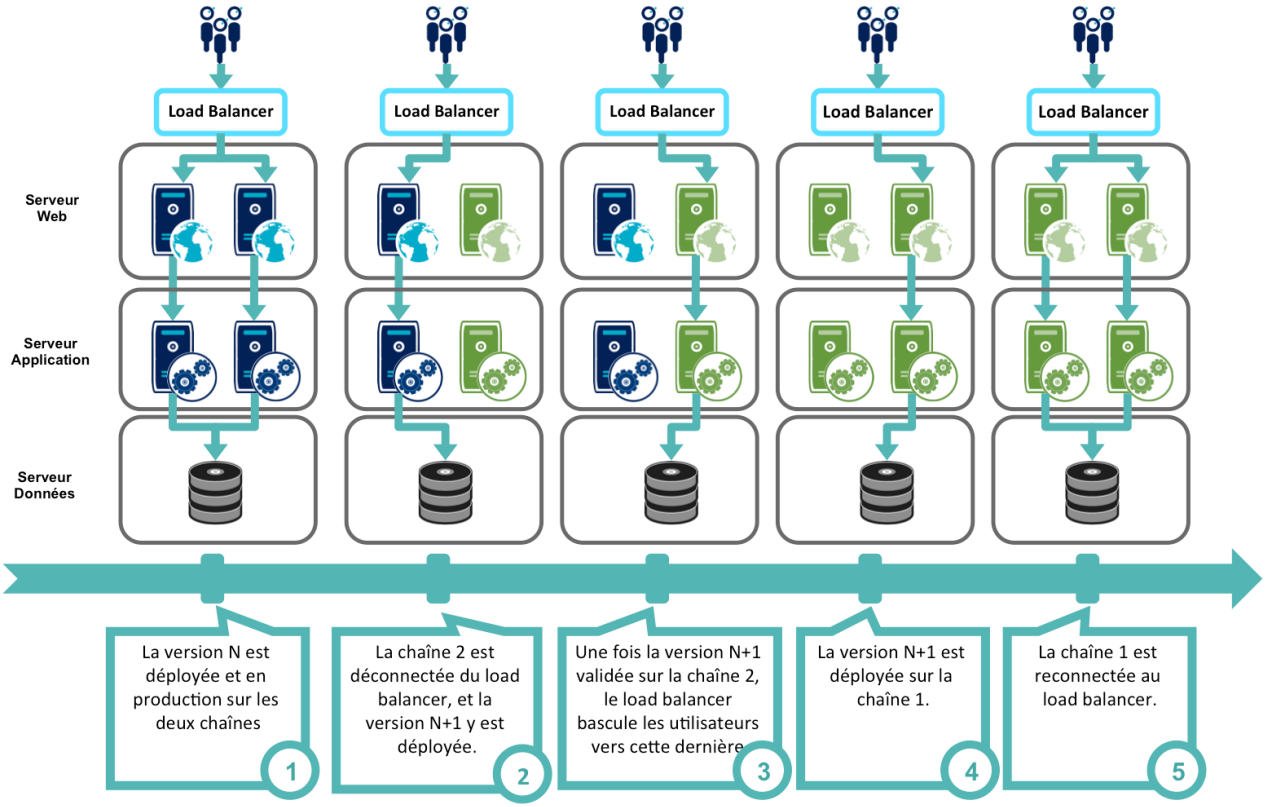
Ce pattern permet de déployer la partie non visible d´une fonctionnalité, en simulant progressivement le traffic qui sera généré par l´utilisation de la fonctionnalité en cible.

L´objectif de ce pattern est pouvoir valider les performances et la scalabilité de la plateforme. En simulant le traffic attendu de manière progressive, on peut préparer et optimiser la plateforme afin que l´ouverture de la fonctionnalité aux utilisateurs finaux se passe dans les meilleurs conditions le jour J.

Mise en oeuvre

L´objectif est d´associer le mécanisme de répartition de charge *(Load Balancer)* à la cinématique de déploiement :

- le *load balancer* déconnecte un des chaînes de production, sur laquelle est déployée la version N+1,
- une fois cette migration effective, le *load balancer* dirige les utilisateurs vers cette chaîne en version N+1,
- il déconnecte l´autre chaîne qui est ensuite mise à jour puis reconnectée à la répartition de charge



Bonnes pratiques

Il existe deux points d'attention dans ce pattern de *ZDD*, le premier concerne les sessions utilisateurs, et le deuxième les changements de schéma de base de données.

Sessions HTTP et répartition de charge

Commençons par un petit point technique : il existe deux façons de gérer les sessions utilisateurs dans le cadre d'une répartition de charge entre plusieurs serveurs applicatifs :

- La première est d'utiliser l'affinité de session,
- La deuxième est d'utiliser des sessions partagées

Affinité de session

Le mécanisme est le suivant :

- Lorsqu'un utilisateur arrive sans session, il est redirigé aléatoirement sur un des serveurs, sur lequel il crée une session
- Le mécanisme de répartition de charge s'assure ensuite que cet utilisateur est ensuite toujours redirigé sur ce même serveur

Ce mécanisme pose des problèmes lors du *Blue/Green Deployment*, puisque les utilisateurs liés au serveur qui sera sorti de la répartition de charge et mis à jour perdront leur session et devront se reconnecter.

Sessions partagées

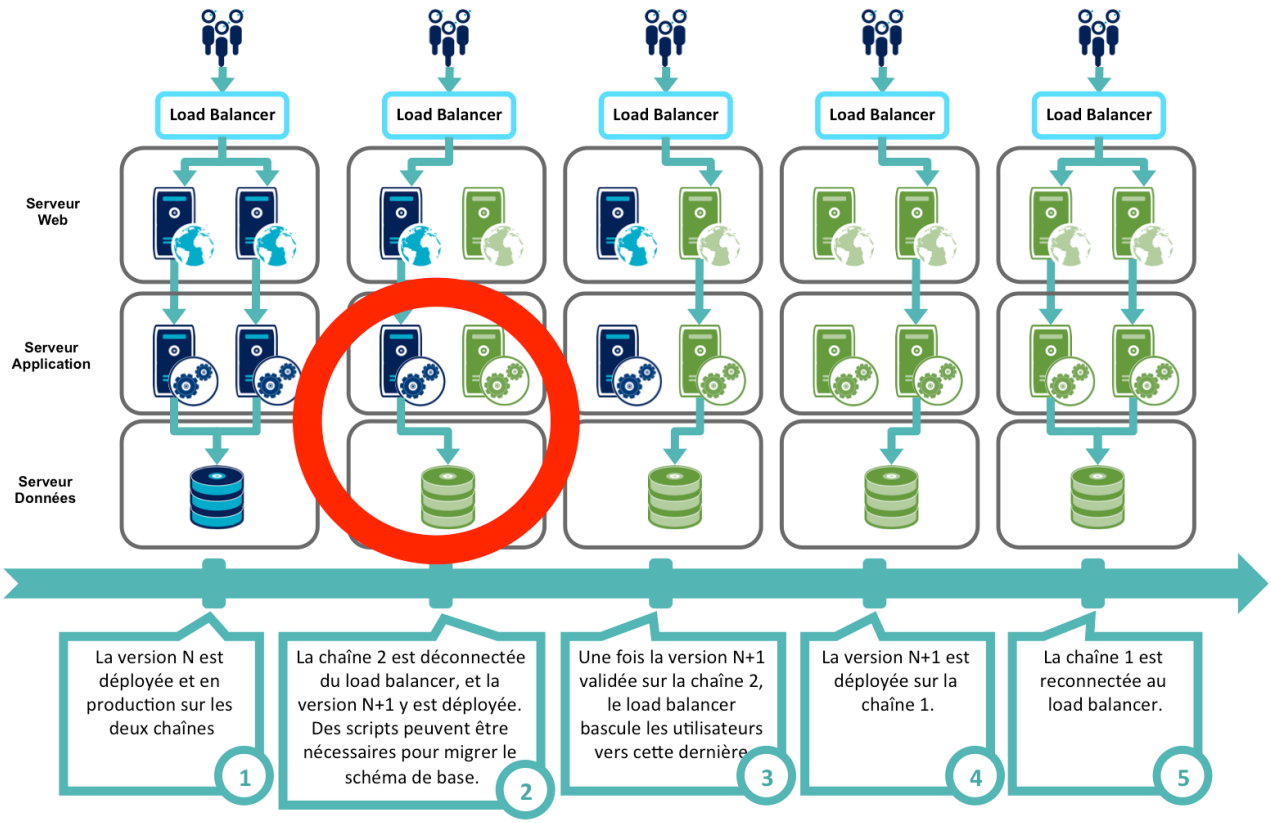
Les serveurs applicatifs partagent un même cache de session, les utilisateurs ne sont liés à aucune des chaînes.

Ce mode de gestion des sessions est évidemment idéal pour le *Blue/Green Deployment*, puisque l'arrêt d'un chaîne n'a aucun impact sur les utilisateurs.

Ce type de mécanisme est fourni en standard chez les principaux serveurs d'application du marché (*Tomcat Cluster, WebSphere ND,* ...). Pour éviter toute adhérence à un produit, il est aussi possible d'implémenter ses propres mécanismes basés sur des outils de cache mémoire distribué, tels *memcached*, ou encore *CouchDB*.

Modifications de schéma de base de données

Reprenons la cinématique du *Blue/Green Deployment*, en ajoutant un contrainte, qui est que la version N+1 de l'application impose des modifications dans le schéma de base de donnés :



L´étape 2 pose un problème : le code applicatif de la version N est **incompatible** avec le schéma de données de la version N+1. Comment s´assurer que la version N fonctionne avec la version N+1 du schéma de base de données ?

Le mot clé est l´**anticipation**, tant du côté de la base de données que du code applicatif :

- Côté base de données, il faut préparer les scripts de migration du schéma :
  - les **scripts d'expansion**, qui vont préparer la phase transitoire pendant laquelle les deux schémas cohabiteront,
  - les **scripts de contraction**, qui permettront de consolider le schéma transitoire pour obtenir le schéma cible
  - les **scripts de rollback**, qui permettront de retourner au schéma initial sans perdre de données en cas de problème

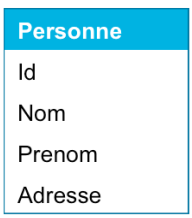
- Côté code, il faut :
  - être capable d´accéder aux différentes versions du schéma de base
  - s´assurer de la **synchronisation** et de la **consistance des données**
  - anticiper la période transitoire, et donc **déployer ce code mixte avant de migrer le schéma de la base**

Vous êtes perdus ? Mettons tout ça en pratique en éclatant une table !

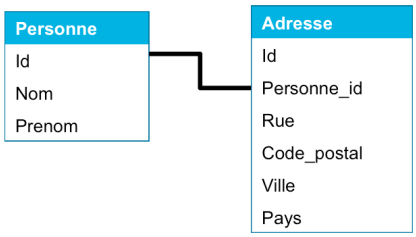
Exemple de migration de schéma

Contexte

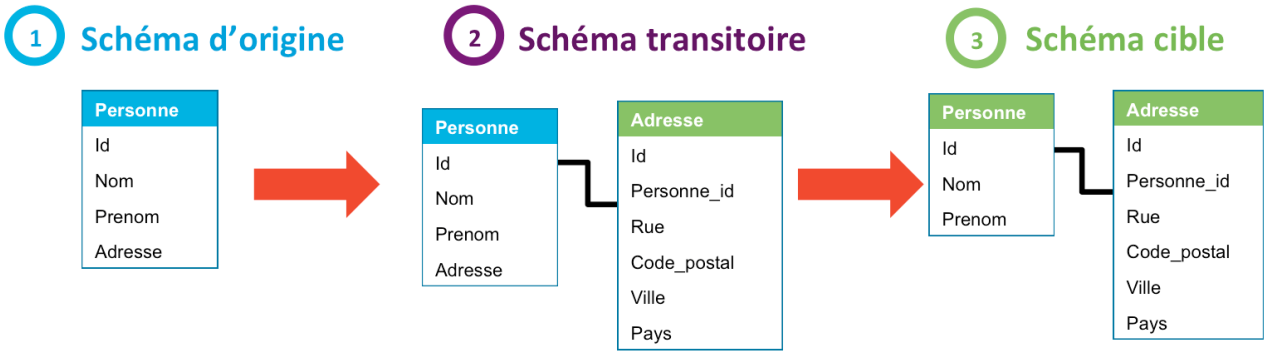
Historiquement, notre application stockait l´adresse d´un utilisateur dans un champ unique de la table *Personne* :



Ce n´est évidemment pas idéal, c´est pourquoi nous allons créer une nouvelle table *Adresse* qui contiendra les différents éléments d´une adresse :



En réalité, cette migration se fera en deux étapes, en passant par un schéma transitoire qui permettra de garantir un retour arrière en cas de problème :



Côté base

Pour permettre cette migration de schéma, trois scripts SQL sont nécessaires :

- Le script d'expansion, qui instancie le schéma transitoire :
  - Création de la table *Adresse*
  - Copie des données de la table *Personne* dans les différents champs de la table *Adresse*

- Le script de contraction, qui migre le schéma transitoire vers le schéma cible :
  - Suppression du champe *Personne.Adresse*

- Le script de rollback, qui permet de revenir au schéma initial :
  - Suppression de la table *Adresse*

Côté code

Le code applicatif doit :

- Gérer indifféremment les différentes versions du schéma
- Maintenir la consistance de ces versions
  - Ex : pendant la phase transitoire, le code doit mettre à jour l'adresse dans les

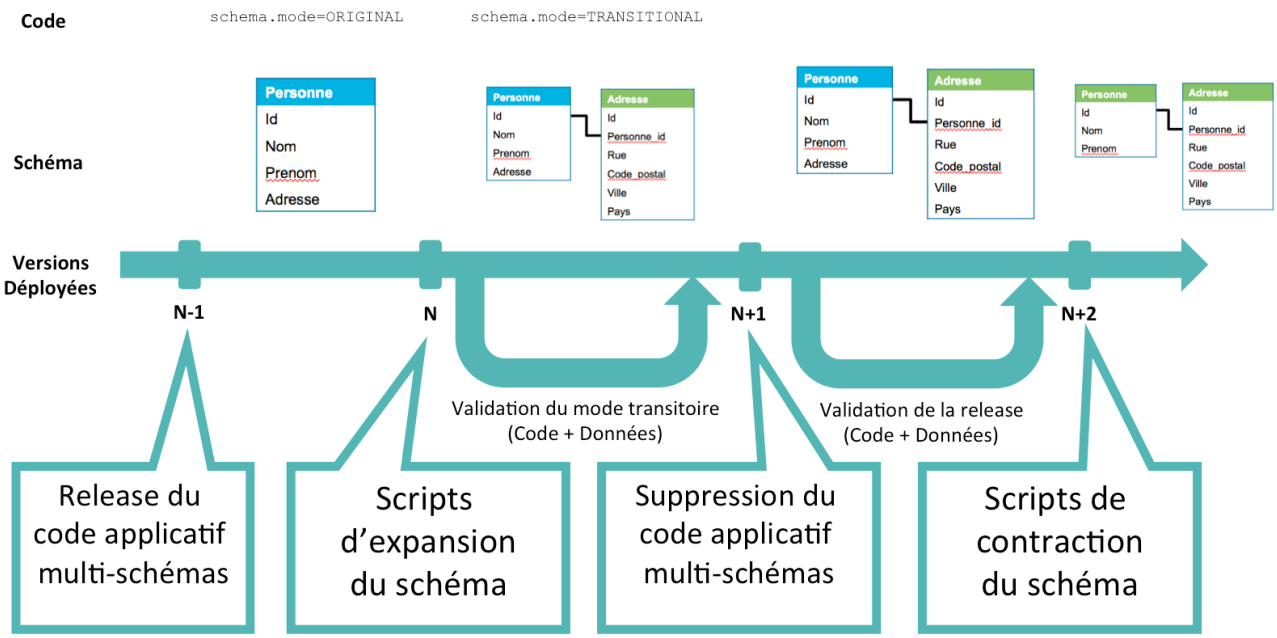
deux tables Personne et Adresse.
- Pour les cas simples (synchronisation de deux colonnes), on peut aussi utiliser des triggers.

Pour permettre au code de gérer les différentes version du schéma de base, la meilleure solution est d'implémenter le Feature Flipping, en déclarant un paramètre qui renseigne le code sur la version du schéma présente :
`schema.mode=[ORIGINAL,TRANSITIONAL]`

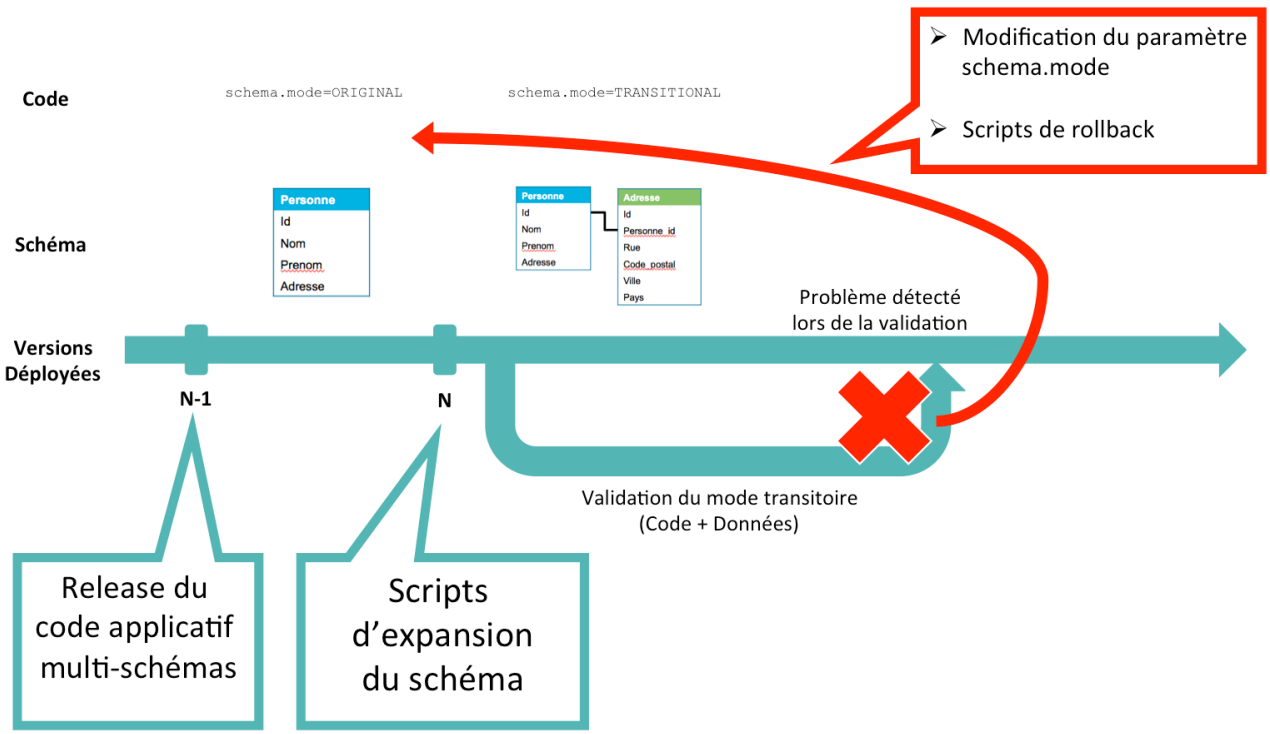
Quant à la consistance des données, le code doit tout simplement écrire dans les deux tables Personne et Adresse, idéalement dans une transaction.

Voyons maintenant comment tout s'enchaîne.

Scénario 1 : migration « sans accroc »



Scénario 2 : retour arrière



Impacts et recommandations

Les migrations de schéma peuvent avoir des impacts négatifs sur les performances de l'application :

- Lors des phases d'expansion et de contraction, qui supposent des créations ou des suppressions d'objets (tables, colonnes), ainsi que leur initialisation
- Durant la période transitoire, qui impose des écritures sur les différents schémas

Pour limiter ces impacts, il est préférable d'éviter le mot clé *ALTER*, qui pose des lock sur les objets impactés. Il est préférable de créer de nouveaux objets (colonnes ou tables). Il est à noter que des outils existent pour limiter les impacts des *ALTER*, on peut citer le *Toolkit* de l'éditeur *Percona*, ou encore l'outil *online schema change* (*OSC*) de *Facebook*, qui permettent de faire évoluer une table par copie et donc sans nécessiter de lock. Ces deux outils concernent la base de données *MySQL*.

Un déploiement qui implique une migration de schéma n'est pas un déploiement comme les autres, il faut donc limiter leur fréquence. Ainsi, chez *Etsy*, les développeurs font une trentaine de déploiements en moyenne par jour, mais un seul déploiement avec modification de schéma a lieu par semaine, le jeudi.

Et pourquoi ne pas aller plus loin, et se passer de schéma ? C'est ce que proposent des bases de données *NoSQL* dites « schema-less », comme l'est par exemple MongoDB.

Chez qui ça fonctionne ?

Les Géants du Web n'interrompent jamais leur service pour le mettre à jour, et ceci pour une raison simple : si on fait un upgrade par mois ou par semaine, on peut très bien avoir une interruption de service de quelques dizaines de minutes, quant on effectue plusieurs dizaines de déploiements par jour, c'est impossible.

Ainsi, les champions du Continuous Deployment que sont *Etsy* et *Flicker* mettent en oeuvre le *Zero Downtime Deployment*.

Et dans le Cloud ?

La société *Netflix* a une vision différente du *Zero Downtime Deployment*. Toute leur infrastructure étant hébergée chez *Amazon*, le paradigme est différent :

- Ils n'ont pas de machine physiques à gérer,
- La création ou la suppression de machines virtuelles est simple et rapide

Pour pouvoir gérer encore plus facilement les déploiements, *Netflix* a développé (et open-sourcé) sa propre surcouche au dessus des outils d'Amazon : *ASGARD*.
Cet outil leur permet de gérer des clusters d'instances, ce qu'*AWS* ne permet pas, mais surtout il fournit une interface de service qui permet à *ASGARD* de s'intégrer avec l'ensemble des outils de *Netflix*.

Le process de déploiement est le suivant :

- Une instance est lancée avec la nouvelle version (*canary machine*) pour faire des premiers tests. l'objectif est de confronter la nouvelle version à des utilisateurs réels. Si tout va bien, on passe au véritable déploiement
- *ASGARD* lance un nouveau cluster avec toutes les instances nécessaires (chez *Netflix* cela peut représenter plusieurs centaines d'instances)
- L'ancien cluster continue de tourner avec l'ancienne version, mais le répartiteur de charge bascule toutes les requêtes sur le nouveau
- On laisse tourner le nouveau cluster un ou deux jours
- Si tout se passe bien, on arrête l'ancien cluster et on recycle ses instances. Le nouveau cluster devient le cluster de prod officiel
- En cas de problème, on redirige les flux sur l'ancien cluster, on arrête le nouveau cluster et on investigue

Grâce à ces outils et à une supervision efficace, Netflix se passe de plate-forme d'homologation. En effet, à part le passage sur le *canary machine*, tout les déploiements se font directement en production.

Et chez moi ?

Tout dépend de la fréquence de vos déploiements et de votre infrastructure de production.

Mais si vous utilisez déjà plusieurs serveurs applicatifs et un mécanisme de répartition de charge, vous avez tout à gagner à faire du Zero Downtime Deployment, même si vous ne déployez une nouvelle version qu'une fois par mois !

Sources

• Martin Fowler, *BlueGreenDeployment*, 10 mars 2010.

• Jez Humble, *Four Principles of Low-Risk Software Releases*, *informIT,* 10 mars 2010.

• Mike Brittain, Etsy, *Continuous Delivery: The Dirty Details*, *goto;* 2 octobre 2012.

• Todd Hoff, *Netflix: Developing, Deploying, And Supporting Software According To The Way Of The Cloud*,*highscalability.com* 12 décembre 2011.

• Jeremy Edberg, Netflix, *Rainmakers: How Netflix Operates Clouds for Maximum Freedom and Agility*, *AWS re:Invent,* 29 novembre 2012.

• Baron Schwartz, Percona, *Zero Downtime Schema Changes*, mai 2012.

• Mark Callaghan, Facebook, *Online Schema Change for MySQL*, 14 septembre 2010.

Suggestion d'articles :

1. **Les Patterns des Grands du Web – Continuous Deployment**
2. **Les Patterns des Grands du Web – DevOps**
3. **Les Patterns des Grands du Web – Design for failure**
4. **Les Patterns des Grands du Web – Minimum Viable Product**
5. **DevOps, de l'intégration continue au déploiement continu**

Cet article a été posté dans Archi & techno, Infrastructure et opérations et taggué Amazon Web

Services, Continuous delivery, déploiement, Déploiement continu, Les géants du web, Les grands du web. Bookmarkez le permalien.
