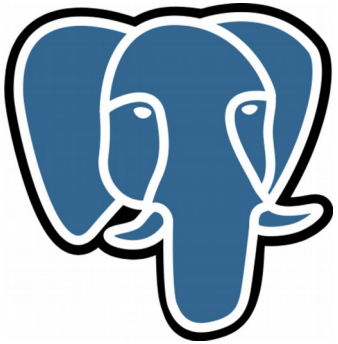
Rudder

Retour d'expérience de PostgreSQL dans Rudder

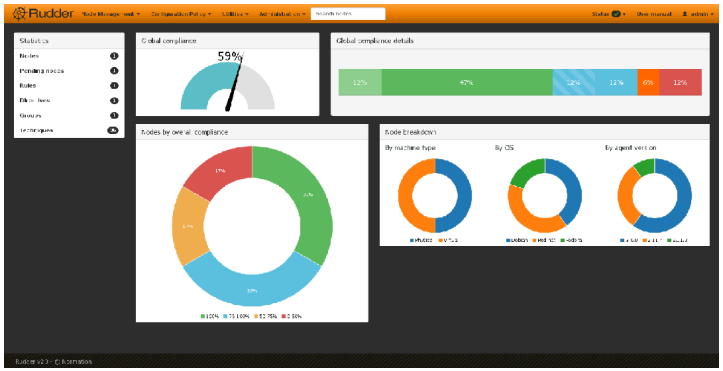
Nicolas CHARLES

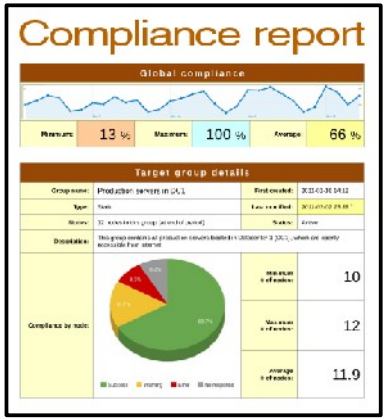
nch@normation.com
@nico_charles



PGDay France
Lille – 31 mai 2016

Solution de gestion et de conformité du socle technique





Accessibilité **Vigilance** **Universalité**

Garantir des fondations solides pour les applicatifs métiers

Présentation de Rudder

Rudder

- Solution open source d'automatisation et de conformité dédiée aux contraintes de production
- But : Faciliter la gestion de configuration et propager son usage
  - *Une interface Web de gestion, avec l'inventaire des systèmes, les configurations appliquées, et la conformité à ces configurations*
  - *Un agent sur chaque système géré, qui compare l'état cible avec l'état du système, corrige si besoin, et remonte les rapports sur l'état.*

Présentation de Rudder

Rudder

- L'agent s’exécute toutes les 5 minutes sur chaque nœud géré
  - Il peut y avoir des milliers d'agents
- L'interface compare les rapports attendus avec les rapports reçu
- Historisation de tous les éléments (nœuds, paramètres de configuration, événements)

Choix technologique

Rudder

- Contrainte: Installation la plus simple possible, sur un maximum de plateformes possible
  - Utilise les dépendances systèmes
- Beaucoup de données structurées
- Quelques données semi-structurées
- PostgreSQL est un choix naturel en 2009
- Les configurations et inventaires sont stockés dans un OpenLDAP (à cause de l'approche hiérarchique des données inventaires)

Choix technologique

Rudder

- PostgreSQL 8.3 est la version disponible dans les distributions cibles de l'époque (Debian 5, SLES 11 SP1)
- En base :
  - Les rapports remontés par les nœuds (données temporaires)
  - Les rapports attendus (semi-temporaire)
  - L'historisation de tous les éléments (persistant)
  - Les event log XML (persistant)

Usage

Rudder

- Les nœuds remontent leurs rapports via syslog, qui sont insérés en base via rsyslog-pgsql
  - La table des rapports fait facilement plus de 150 Go
- L'application calcule, lors de la génération des règles de configurations, les rapports attendus (stockés dans des tables structurés avec FOREIGN KEYS)
- Le calcul de conformité est une comparaison entre les rapports attendus et les rapports reçus (basés sur les ID de configurations, ID et version de règles, composants de configurations)
  - (C'est des gros SELECT et JOIN imbriqués)

Usage

Rudder

- Chaque action dans l'interface crée un événement, stocké en base
  - Du XML, avec un format versionné (des nouveaux types d’événements apparaissent régulièrement)
  - Recherche en XPATH pour afficher dans l'interface
  - Mise à jour via des scripts lors des montées de version

Usage

Rudder

- Evolution régulière du schéma de stockage
  - Nouvelles informations à stocker
  - Changement du format pour avoir des rapports de conformité plus précis
  - Mise à jour via des scripts lors des montées de version

Retour sur l'usage

Rudder

- Ça a fonctionné directement !
  - Aucun soucis d'UTF-8 (pas de surprise de langue suédoise par défaut)
  - Pas de problèmes de timezone
  - Evolution facile du schéma entre chaque versions
    - Excellent flexibilité et versatilité
  - En Scala, PostgreSQL est un standard
    - Toutes les lib de gestion de base de données reconnaissent les structures natives (XML, Array)

Retour sur l'usage

Rudder

- Ça a fonctionné directement !
  - Pas de problèmes sur des requêtes complexes
    - Nous n'avons jamais été bridé dans notre expressivité
  - Des performances toutes à fait correctes sans customization particulières

Retour sur l'usage

Rudder

- Très bonne robustesse
  - Même en cas de remplissage de tous l'espace disque et crash, les bases n'ont jamais été totalement corrompues
    - Pertes des derniers rapports uniquement
    - Réparation facile en suivant les documentations

Retour sur l'usage

Rudder

- Tests unitaires
  - Avec des tables temporaires qui ne polluent pas les données déjà présentes
    - Bien penser à avoir un pool de connexion d'une seule connexion, sinon les données ou tables disparaissent en cours de test..

Performances

Rudder

- PostgreSQL est très accommodant
  - Par défaut, il fera ce qu'il faut
    - Optimisation automatique des indexes en fonction des requêtes
    - Performances tout à fait corrects
  - Ça juste marche (quand la volumétrie reste limité)

- EXPLAIN ANALYZE
  - Permet de comprendre comment la requête va être traité
  - Identifie les points de contentions
  - Amélioration des requêtes très facile
    - Mais il ne faut pas faire trop de fois de suite la même requête pour l'évaluer, sinon elle passe dans le cache
  - Mais besoin de vrais jeux de données
    - On va revenir sur ce point

https://explain.depesz.com

Nos problèmes

Rudder

- Quand la volumétrie a augmenté, on a eu des problèmes liés à nos usages
  - Les rapports sont insérés et supprimés (durée de vie de 4 jours)

    → Index qui bloatent en 8.x, explosion de l'espace disque nécessaire et chute des perfs régulières (en 2013)

    → VACUUM FULL, voir REINDEX régulier sur les tables de rapports
  - Le problème était déjà corrigé en version 9.x
  - Arrêt du support de PostgreSQL 8.x à partir de Rudder 3.0 (2015) : problème résolu

Nos problèmes

Rudder

- Nos clefs primaires n'allaient que jusqu’à 2 milliards
  - Les ints ne sont que sont que sur 32 bits
  - *Oups*…

    → Limite à 70 jours d'activité avec 1000 nœuds.
  - Migration sur des id en bigint (2014) : problème résolu

Nos problèmes

Rudder

- Quand la volumétrie a beaucoup augmenté, on a eu d'autres problèmes liés à nos usages
  - Les rapports attendus deviennent trop nombreux
    - On n'avait pas prévu de nettoyage des vieux rapports, parce que PostgreSQL se débrouillait même avec des grosses tables
    - Mais au bout d'un moment, trop, c'est trop
    - Mise en place d'une purge des vieux rapports attendus (en 2015) : problème résolu

L'utilisation sur une vraie Prod

Rudder

- Trois possibilités :
  - Soit on héberge le logiciel chez nous (en mode SaaS) et on sait ce que font vraiment les utilisateurs,
  - Soit on doit reproduire les infrastructures du client sur nos plateformes de tests,
  - Soit on doit dealer avec les utilisateurs pour extraire des infos et exécuter des requêtes en local.

L'utilisation sur une vraie Prod

Rudder

- Notre solution n'est pas hébergés en SaaS
- Nos plateformes de tests sont assez limitées
  - Nos clients ont des productions avec plus de 5000 machines, avec des installations depuis plusieurs années
    - Il est difficile d'extrapoler depuis nos tests le comportement sur ces plateformes
  - Les bases de données des clients ne sont souvent pas connectées à Internet
    - Pas d'accès distant possible

L'utilisation sur une vraie Prod

Rudder

- Un gros utilisateur a accepté de faire tourner son PostgreSQL en loggant toutes les requêtes
  - Et nous a envoyé les Go de logs résultants :
    - 10 millions de query
    - 3000 query/secondes en peak
- Analyse via PgBadger de ces logs
  - Une requête fréquente prend 48 secondes sur une énorme infra
  - *Oups* (encore)
  - Ça n'était pas du tout visible sur des infras de moins de 1000 machines

L'utilisation sur une vraie Prod

Rudder

- Requête qui recherche dans un array les clefs de configurations
  - Retourne les rapports attendus
  - Avec un generate_subscripts et des jointures

```
SELECT E.pkid,
       NNN.nodeid,
       NNN.nodeconfigids
FROM expectedreports E
INNER JOIN
  (SELECT NN.nodejoinkey,
          NN.nodeid,
          NN.nodeconfigids
   FROM
     (SELECT N.nodejoinkey,
             N.nodeid,
             N.nodeconfigids,
             generate_subscripts (N.nodeconfigids, 1) AS v
      FROM expectedreportsnodes N) AS NN
   WHERE NN.nodeconfigids [ v ] IN ('value1',
                                    'value2',
                                    'value3')) AS NNN ON E.nodejoinkey = NNN.nodejoinkey
```

https://explain.depesz.com

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.722	12,018.918	↑ 955.2	230	1	→ Hash Join (cost=1,201.200..230,348.640 rows=219,689 width=92) (actual time=734.703..12,018.918 rows=230 loops=1) Hash Cond: (two.six = lima.six)
2.	11,976.309	11,976.309	↑ 392.8	80	1	→ Seq Scan on hotel two (cost=0.000..225,379.500 rows=31,421 width=92) (actual time=692.679..11,976.309 rows=80 loops=1) Filter: (tango && '{-729463641,-1360796549,-1647020772,-1069407658,-169478327,414076556,-1160380758,587719827,2003792667,-1527719623,-2002282
3.	18.895	41.887	↓ 1.0	24,396	1	→ Hash (cost=906.090..906.090 rows=23,609 width=8) (actual time=41.887..41.887 rows=24,396 loops=1) Buckets: 4096 Batches: 1 Memory Usage: 953kB
4.	22.992	22.992	↓ 1.0	24,396	1	→ Seq Scan on yankee lima (cost=0.000..906.090 rows=23,609 width=8) (actual time=0.010..22.992 rows=24,396 loops=1)

- Visualisation claire des points de contentions
- Anonymisation des données pour partage

L'utilisation sur une vraie Prod

Rudder

- Remplacé par une intersection avec &&
  - Et un index GIN sur l'array

```
SELECT E.pkid,
       NNN.nodeid,
       NNN.nodeconfigids
FROM expectedreports E
INNER JOIN
  ( SELECT N.nodejoinkey,
           N.nodeid,
           N.nodeconfigids
   FROM expectedreportsnodes N
   WHERE (nodeconfigids && '{"value1", "value2", "value3"}') ) AS NNN ON E.nodejoinkey =
 NNN.nodejoinkey;
```

- Résultat : 49ms chez l'utilisateur

Documentation

Rudder

- La documentation est parfois inégale
  - Excellente sur la majorité des points
    - Notamment sur l'administration générale
  - Pas forcement claire sur ce qui est performant ou non (par exemple, generate_subscript et &&)
  - Un peu sibylline sur le XML et le JSON

Configuration mémoire

Rudder

- La configuration mémoire de PostgreSQL par défaut est aberrante
  - Pour des systèmes tellement petits qu'ils n'existent plus
- On doit donc augmenter des paramètres
  - Les conseils sont très souvent divergeant
    - Augmenter le shared_buffer ou laisser le kernel gérer ?
    - Work_mem or not work_mem ?
    - checkpoint_segments
      - je fais des calculs, mais je ne suis pas certains de leur pertinence

Evolution prévue de nos usages

Rudder

- Rudder est on-premise, donc on a un objectif fort de simplicité
  - L’intérêt d'OpenLDAP face à PostgreSQL pour notre usage n'est plus évident
  - Stockage en JSON dans PostgreSQL des données d'inventaires et de configuration
    - Attendre que la version 9.3 soit bien démocratisée

Evolution prévue de nos usages

Rudder

- Mieux gérer les données selon leur cycle de vie
  - Consommer et traiter les rapports remontés plutôt que de les stocker dans une base directement
- Implémenter efficacement les cas de traitement asynchrone (pour l'affichage par exemple)
  - Evolution qui va de la base de données à l'interface dans le navigateur

Evolution prévue de nos usages

Rudder

- Réplication des données
  - Pour l'instant, ce sont nos utilisateurs qui traitent directement (et ça nous arrange)
  - On voit bien que le master-master ne sera pas évident dans les vieilles versions
- On joue la montre
  - Beaucoup de travail dans PostgreSQL dans ce sens et on espère que ça sera facile quand on aura à se pencher sérieusement dessus

Conclusions

Rudder

- Robuste
- Accommodant
  - PostgreSQL « apprend » des usages, pour fournir des bonnes performances, même après avoir changé toute le schéma de la base
- Tous les problèmes rencontrés étaient de notre responsabilité
  - Ou alors déjà fixés dans des versions plus récentes
- L'évolution générale du produit est super

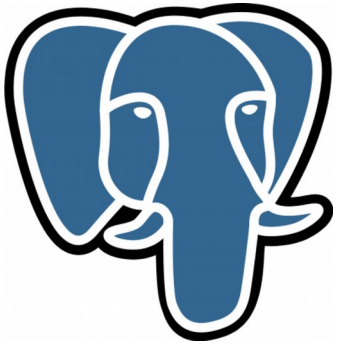
« L'industrialisation de PostgreSQL est excellente, rarement égalée même par des logiciels propriétaire "enterprise" »

Rudder

Retour d'expérience de PostgreSQL dans Rudder

Nicolas CHARLES

nch@normation.com
@nico_charles



PGDay France
Lille – 31 mai 2016