TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

Gülçin Yıldırım 156398IASM

NEAR-ZERO DOWNTIME AUTOMATED UPGRADES OF POSTGRESQL CLUSTERS IN CLOUD

Master's thesis

Supervisor: Tarmo Robal
PhD

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Gülçin Yıldırım 156398IASM

KIIRE POSTGRESQLI KLASTRITE AUTOMATISEERITUD UUENDAMINE PILVANDMETÖÖTLUSE PLATVORMIL

Magistritöö

Juhendaja: Tarmo Robal
PhD

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Gülçin Yıldırım

08.05.2017

Abstract

One of the biggest challenges regarding database upgrades is how to reduce the required downtime of a system when the upgrade is happening. It is not always possible to have a maintenance window for upgrade operations due to the unwanted effects of system outages such as financial costs, revenue loss, damaging the business reputation, the risk of not meeting Service Level Agreements (SLAs). The impact of the downtime may vary considerably across industries, also depend on the business size. In modern systems, zero-downtime upgrades is a big need and in fact, can be achieved through the advancements in IT automation, configuration management, and orchestration methodologies. Additionally, the emergence of Cloud Computing technologies enables applying software solutions at a scale, in cost and time-efficient ways.

The aim of this thesis is to provide an automated software platform to achieve near-zero downtime upgrades of PostgreSQL clusters in cloud environments. The proposed method uses logical replication as means of upgrading to the new PostgreSQL version. The platform utilizes open-source logical replication tool, Pglogical, to replicate the changes to the new server. The application switch between the old and the new server is handled gracefully by using a PostgreSQL-protocol aware connection proxy, PgBouncer.

The experimental results show that this method could be viable approach for achieving minimal system interruption with less than 10 seconds of downtime.

This thesis is written in English and is 60 pages long, including 7 chapters, 14 figures and 3 tables.

Annotatsioon
Kiire PostgreSQLi klastrite automatiseeritud uuendamine pilvandmetöötluse platvormil

Üks suurimaid probleeme seoses andmebaasi versiooniuuendusega on see, kuidas vähendada süsteemi nõutavat maasolekuaega versiooniuuendamise ajal. Alati ei ole võimalik uuenduste jaoks garanteerida aega hoolduseks teatud soovimatute efektide tõttu, näiteks rahaliste kulude, tulude kaotamise, ärimaine kahjustamise, teenusetaseme kokkulepete (SLA-de) riskist tuleneva soovimatu mõju tõttu. Kaasaegsetes süsteemides on vajalik tagada versiooniuuendused minimaalse maasolekuajaga, mida on võimalik saavutada IT protsesside automatiseerimise, konfiguratsioonihalduse ja orkestratsiooni metoodikate abil. Lisaks võimaldab pilvandmetöötluse tehnoloogiate arendus tarkvara lahenduste rakendamist skaleerida ning ajaliselt ja maksuvuselt optimeeritumaks muuta.

Antud väitekirja eesmärk on luua automatiseeritud tarkvaraplatvorm, et saavutada PostgreSQL-i klastrite liginullmaasolekuaeg pilvekeskkonnas. Selleks uuriti PostgreSQL-i ja muude seostuvate andmebaaside versioonilahendusi. Olemasolevad sisseehitatud andmebaasi uuendamise meetodid PostgreSQL-i jaoks ei olnud antud eesmärgi jaoks sobivad. Seetõttu uuris antud lõputöö autor loogilist replikatsiooni PostgreSQL-i baasil. Kasutades avatud lähtekoodiga Pglogical laiendust pakutud uuendusmeetodi baasina, rakendas autor automatiseeritud PostgreSQL-i klastrite uuendamise tööriista Pglupgrade, mida kasutatakse muudatuste kopeerimiseks uude serverisse. Vana ja uue serveri vahelist rakendusrežiimi hallatakse läbi PgBouncer proksi. Uuendusprotsessi organiseerimiseks kasutati Ansible IT automatiseerimise vahendit.

Pglupgrade tööriistaga välja töötatud uuendusmeetodi hindamiseks viidi läbi kaks juhtumiuuringut. Esimene juhtumisuuring oli keskendunud väikesele klastritele, mis loodi kõrge käideldavuse tõttu. Pglupgrade oli ainus meetod, mis ei seganud rakendust. Rakendus käsitlses 3-sekundilist maasolekuaega kui viivitust. Teine juhtumiuuring viidi läbi suurema klastriga, et hajutada päringuid süsteemi koormuse tasakaalustamiseks. Pglupgrade lähenemine ületas olemasolevaid meetodeid ka teises eksperimendis, saavutades üleüldise miinimumi 5-sekundilise primaarse maasolekuajaga, seejuures põhjustamata katkestusi töötavas rakenduses, nagu oli ka esimese juhtumisuuringu korral.

Antud väitekiri näitas, kuidas saab andmebaasi klastreid uuendada minimaalse maasolekuajaga. Autor on näidanud, et loogilise replikatsiooni ja proksiühenduse abil on võimalik muuta rakendused ja nende kasutajad teadmatuks, et andmebaasi versiooni uuendatakse kõigest marginaalse jõudluse vähenemisega. Pglupgrade töövahend on demonstreerinud selles töös kirjeldatud ideede praktilist rakendamist ja osutunud meetodi kasutatavusele, saavutades minimaalse süsteemi katkestuse vähem kui 10 sekundilise maasolekuajaga.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 60 leheküljel, 7 peatükki, 14 joonist, 3 tabelit.

List of abbreviations and terms

ACID	Atomicity, Consistency, Isolation and Durability
ADDM	Automatic Database Diagnostic Monitor
AMI	Amazon Machine Images
API	Application Programming Interface
AWR	Automatic Workload Repository
AWS	Amazon Web Service
BASE	Basically Available, Soft state and Eventual consistency
BSON	Binary JSON
CAP	Consistency, Availability, Partition-tolerance
COMFORT	Comfortable Performance Tuning
CPU	Central Processing Unit
CSV	Comma-separated values
DB	Database
DBMS	Database Management System
DBUA	Database Upgrade Assistant
DDL	Data Definition Language
DMV	Dynamic Management Views
DNS	Domain Name System
DSL	Domain Specific Languages
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
EPP	Embedded Puppet
ERB	Embedded Ruby
EU	European Union
FDW	Foreign Data Wrapper
FOSDEM	Free and Open Source Developers' European Meeting
GUI	Graphical User Interface
HA	High Availability
IaaS	Infrastructure as a Service
IBM	International Business Machines
ICT	Information and Communications Technology
IP	Internet Protocol

IT	Information Technology
JSON	JavaScript Object Notation
MODA	Automated Test Generation for Database Applications via Mock Objects
MSSQL	Microsoft SQL Server
MVCC	Multiversion Concurrency Control
NASA	National Aeronautical Space Agency
NIST	National Institute of Standards and Technology
NoSQL	Not only SQL
OLTP	Online Transaction Processing
OUI	Oracle Universal Installer
PaaS	Platform as a Service
PC	Personal Computer
PRISM	Automating Database Schema Evolution in Information System Upgrades
RAM	Random Access Memory
RCU	Repository Creation Utility
S3	Simple Storage Service
SaaS	Software as a Service
SAT	Boolean Satisfiability Problem
SLA	Service Level Agreement
SQL	Structured Query Language
SQLCM	SQL Continuous Monitoring
SQS	Simple Queue Service
SSH	Secure Shell
TCP	Transmission Control Protocol
TOPDB	Top Database Index
TPC-B	Transaction Processing Performance Council Benchmark B
TPS	Transactions Per Second
VPC	Virtual Private Cloud
WAL	Write Ahead Log
XML	Extensible Markup Language
YAML	Yet Another Markup Language

Table of Contents

List of Figures

List of Tables

1 Introduction

A database can be defined as an organized collection of data. A computer software application which is responsible for the safe and efficient storage as well as the easy retrieval of the data in a database is called a Database Management System (or simply a database system or a DBMS [1]).

Many software applications use database systems to store and process information. For instance, banks use databases to store account information and balances of their customers, social networks use databases to store user profiles, contacts and messages. Even desktop applications that do not need to process much data like web browsers (i.e Firefox or Chrome), use databases to store session information and browsing history. Nowadays, companies consider customer data is one of the most important assets for their businesses [2] and as a result, database systems are highly important in IT ecosystem.

Modern companies aim at global markets and they maintain their services in many countries all around the world. This flexibility comes with a cost; high availability in multiple regions is a must for companies to continue their business successfully. That is why this thesis illustrates the usage of cloud platforms instead of bare-metal servers. Cloud platforms allow to scale in multiple regions, resize the instances, terminate or add more instances, and scale in a dynamic manner instead of static scalability nature of traditional servers.

Given the importance of database systems for the businesses, reliability, stability, and security of the databases are also equally important. Outdated databases are vulnerable to attacks and keeping the databases in their current versions would help to reduce the security risks. An obsolete version of a database has many disadvantages such as lack of technical support, incompatibility with hardware and software, missing enhanced features, bug fixes, performance improvements and security patches. Therefore, regular database upgrades should be a policy of system designers to benefit from all the new features and avoid the risks of running the systems on outdated databases.

Even though there are clear advantages for database upgrades, it still means downtime for many people and companies struggle to set maintenance windows for major version upgrades. Moreover, thanks to fast growth of data volumes in recent years, few companies can manage to complete the upgrade within the possible maintenance window. On top of this, there are many businesses that would not allow a maintenance for database upgrades at all (i.e. payment systems, online banking and money transfer companies, nuclear plants, space technologies, telecoms). Both business owners and the users prefer not to experience downtime, this leads to efforts to avoid downtime as much as possible. The impact of taking down a system that has thousands or maybe more than millions of users for a long time means losing revenue, reputation and sometimes even customers. In modern systems, zero downtime upgrades is a big need and in fact, can be achieved.

There are four main possible approaches to database upgrades:

- The first approach would be for databases to keep their storage format same or at least compatible across versions. However, this is hard to guarantee long term as new features might require changes in how data is stored or add more metadata information to work properly. Also, performance is often improved by optimizing the data structures.
- The second approach is to make a logical copy (dump) of the old server and loading it into the new server. This is the most traditional approach which requires the old server to not receive any updates during the process and results in prolonged downtimes of hours or even days on large databases.
- The third option is to convert data from old format to new one. This can either be done on the fly while the new system is running, but incurs performance penalty which is hard to predict as it depends on data access patterns, or it can be done offline while the servers are down, again incurring prolonged downtimes (although often shorter than the second method).
- The fourth method is to use logical dump for saving and restoring the database while capturing the changes happening in meantime and logically replicating them to the new database once the initial restore has finished. This method requires orchestration of several components but also decreases the amount of time the database cannot respond to queries.

The example of the approaches listed above can be seen in popular relational database management system PostgreSQL. For example, PostgreSQL minor releases, which do not contain new features but only fixes, do not change the existing data format and fits the first approach. For the second approach, PostgreSQL provides tools called pg_dump and pg_restore which do the logical backup and restore. There is also a contrib module[1] called pg_upgrade[2] which does offline (the servers are not running) conversion of the old data directory to the new one. For the fourth approach, there are third party trigger-based solutions such as Slony[3] for upgrading but it also has some caveats which are covered in the comparison of all existing methods in Chapter 3.2.2.

Each major release of PostgreSQL comes with a wide set of features, therefore skipping version upgrades might create a bigger gap in terms of existing feature set of the applications and the one it will get after the upgrade. When decision makers of the companies calculate the risks of the upgrade and its impact on the business, they might even be forced to continue running a version of PostgreSQL that is no longer officially supported and has known data corruption or security problems. Result of a recent survey [3] published in 2015 and more than 200 enterprises are surveyed, shows that more than 60% of enterprises have deferred applying security patches to their databases because of concerns over downtime. Avoiding major release upgrades usually affects the developers

[1] https://www.postgresql.org/docs/current/static/contrib.html
[2] https://www.postgresql.org/docs/current/static/pgupgrade.html
[3] http://www.slony.info/

who are missing the new features and the performance improvements. This, in turn, affects the man hours which are needed for continued development of the application, because the developers cannot use the new features of the database. The additional resources, both in terms of hardware and developer time, spent on performance improvements could often be saved by simply upgrading to the new major version of the database software. PostgreSQL has a wide feature set which comes with every major release and companies can benefit greatly from doing regular upgrades.

Another consideration to take into account while upgrading the database is that there is very seldom only single server. Both for reasons of availability (having another server to switch application to when the current one fails) and read scalability (providing extra hardware resources to improve the performance of database reads), the physical streaming replication[4] is often used to create cluster of one primary and several replica servers. Traditional methods of upgrade can only upgrade the primary server while the replica servers have to be rebuilt afterward. This leads to additional problems with both cluster availability and capacity, hence effectively increasing the perceived downtime of the database from the point of both applications and users.

An additional challenge for any kind of system change is testing. It is important to test the new database system version before the switchover is completed. Logical replication allows replicating while the system is up-and-running and testing effort can be handled in the meantime. Software automation allows running tests to ensure the validity of the process while being sure that the steps in production will be same as the ones during testing, due to the removal of the human factor. The automation combined with the use of replication also allows rolling the system back to the previous state in case of unexpected problems.

The aim of this thesis is to describe and implement a solution which focuses on solving the problems discussed above. Main goals of the described solution are to minimize the required downtime, ensure business continuity, and improve predictability of the PostgreSQL major version upgrade process in the cloud. The solution is built on three main ideas. These are, automating the upgrade process, using logical replication to copy data from old server to the new one while the old one is still actively serving requests, and finally using connection proxy to make the switch to new server transparent to the applications.

To develop the platform, Ansible is chosen as configuration management and automation tool for the purpose of the orchestration of the upgrade process. Ansible has modules to support many cloud vendors which make the study applicable to the other cloud platforms with minor changes in the application code. This removes the vendor-locking issue related with cloud providers. The proposed solution is designed to use replication to a new instance as means of upgrading. The chosen method for doing the replication is to use the

[4] https://www.postgresql.org/docs/9.6/static/protocol-replication.html

power of logical decoding which is available in PostgreSQL versions 9.4 and later. Logical decoding is preferred over trigger-based solutions, considering logical decoding has a lower impact on the system. However, logical decoding only provides API to consume raw changes from the database, in order to replicate the changes to the new server, a tool which implements the actual replication is still needed. A third party open source tool called Pglogical [4] is used for this purpose. To ease the transition of application between old and new server, a PostgreSQL-protocol aware connection proxy, called PgBouncer [5] is used. The author utilizes this tool to make this transition transparent to any application using the database server.

The outline for this thesis is as follows. In Chapter 2, the current state of the automated systems is researched by their relevance to the thesis. Evolution of the automated software solutions are evaluated and the areas that need to be improved are discussed. The background of the technologies that has been used in the thesis presented in Chapter 3, more specifically Relational Database Management Systems in the example of PostgreSQL, Cloud Computing as a concept, the software-based management of Cloud infrastructures and configuration management for software deployment onto such infrastructures. Chapter 4 4 Automated Upgrades in PostgreSQLcovers the conceptual design of the Automated Cluster Upgrades Platform from a high-level perspective, i.e. the interactions between different components of the system and the user’s interaction with the system. Moreover, the actual implementation of the Automated Cluster Upgrades Platform is detailed in the same chapter. Chapter 5 analyzes the impact of PostgreSQL upgrades. Chapter 6 presents two case studies to evaluate the performance of the platform as well as the effort needed to deploy and execute distributed applications with it. Finally, Chapter 7 summarizes the thesis.

2 Software Automation of ICT

This section surveys related state-of-the-art work in the field of automated software and system design.

The dictionary[5] defines automation as “the techniques and equipment used to achieve automatic operation or control.” The author believes that the main motivation behind automating a process or a system to operate automatically is reducing the risks associated with the human factor and increasing the reproducibility of the outcome. Besides, automated processes will often be faster than the same task performed manually that will result in better efficiency and lower operating costs.

While deciding what to automate in a system, the obvious rule is to consider automating processes that are expected to be repeated frequently throughout the system life cycle. The value of automating a process is higher if it will be repeated more often. Another thing to remember is that automating a system (or a subsystem) should not be just keeping the old, inefficient processes, but re-engineer and change them fundamentally. For example, in 1995, when Wal-Mart decided to optimize the cost caused by unnecessary distribution steps, they chose to redesign the entire supply chain, instead of just improving their existing distribution mechanism. They built (along with software companies) an enterprise-wide system that directly connects all retail locations, distribution warehouses, and major suppliers. The elimination of unnecessary distribution steps allowed them to provide value to customers by reducing costs [6]. Hereby, the goal of automation should be a dramatic change and not just incremental improvement.

After the identification of which processes to automate and which ones to modify or eliminate, a research is required how to automate those processes. When the objectives are listed, the existing processes are mapped, measured, analyzed, and benchmarked, these efforts are combined to develop a new business process [7]. In this stage, automation plan should also cover tests and rollout plans. However, before executing the automation plan, the estimated automation effort versus the risk of performing manual procedures should be evaluated. Like any other (mainly business) decision, cost-benefit analysis of the automation should be performed.

If the project is close to ending of its timeline, there is possibly very little benefit to automating a process. On the other hand, if planned properly, an early planning of a will-be-frequent process can save a lot of time and help to reduce the cost of the project. Obtaining the greatest benefit from automation, companies should close the gaps between different teams of people who come from different backgrounds and have a different area of expertise. Capabilities of IT could be used to create and support cross-functional teams instead of individuals working in isolated departments [7].

[5] http://www.thefreedictionary.com/automation

Information technologies continue to evolve and the digital world requires eliminating manual processes to evolve even faster, time to market and reliability of the operations are crucial for businesses. Manual tasks are known to be error-prone, hence reducing manual operations allow shifting human force to more advanced tasks rather than simple, repetitive, monotonous and time-consuming operations. IT systems consist of many subsystems working together in a harmony, an output of a subsystem can be the input of another subsystem (or subsystems), optimizing one subsystem can dramatically affect related subsystems and the whole system altogether. Automation of the processes, tasks, and systems eliminates tedious work, increase the confidence in human resources, unexpected results and possible failures can be minimized.

Understanding reasons behind of the automation might help to understand how automated systems are helpful, where they apply in software technologies, and why they are preferred. To begin with, repeatability of the automated processes carries a great value in information technologies. For instance, a set of manual operations executed by different teams have the risk of uncertainty involved in the manual processes, but an automated script could guarantee that the same instructions will be executed in the same order each time the same script is run independently of who runs the script. Furthermore, using the same example, scripts are more reliable because they reduce the chances of human error.

Moreover, automated tasks are often faster than the same tasks performed manually, results with greater efficiency in automation. NASA used Ansible for automating their migration to cloud-based environment from a traditional hardware-based data center[6]. As a result of automation, NASA has increased their overall efficiency. For example, updating nasa.gov went from over 1 hour to under 5 minutes, patching updates went from a multi-day process to 45 minutes, and application stack set up from 1-2 hours to under 10 minutes per stack. Testing and versioning are the other common reasons why automated systems are preferable in IT. Once processes are automated, testing of scripted processes undergoes throughout the development cycle. Unlike manual testing, scripts are proven set of repeatable processes which make them more mature as the project progresses. Also, automated systems can be placed under version control system, hence making them more trackable as any other piece of code in the system, this eliminates the risks come with manual versioning of individuals and reduces human errors.

Nowadays, Information Technologies are converting into more automated systems and processes such as self-managing, self-monitoring, self-healing, self-optimizing, self-protecting and self-tuning systems. Academic literature research shows that database administration and management field is no different than other fields of software related automation practices. A good example for automation in database administration world is an attempt to automate performance tuning process of databases. The COMFORT (Comfortable Performance Tuning) project [8] studied a prototype architecture to provide an approach to create a self-tuning database system that can dynamically adapt the system

[6] https://www.ansible.com/hubfs/pdf/Ansible-Case-Study-NASA.pdf?t=1481315114902

parameters to the evolving workload. In real-life software applications, system load is usually not static and has peak hours as well as quiet times through a day. Therefore, modern systems require flexible solutions for handling these fluctuations in load by reacting towards it by creating responsive subsystems such as self-tuning databases as prototyped in COMFORT paper [8], or auto-scaling mechanisms as provided in AWS Auto Scaling[7] feature.

Database Management Systems have been providing automated solutions to ease the database administration and management processes within their automation capabilities. Three commercial products are to exemplify: First, Oracle Self-Managing Database [9] framework offers tools such as Automatic Workload Repository (AWR) which gathers system data so that Automatic Database Diagnostic Monitor (ADDM) could analyze the collected data and make recommendations for the best interest of the system. Automatic Shared Memory Management feature automates the management of shared memory used by an Oracle instance, according to the demands of the workload that is collected by AWR and examined by ADDM.

Second, IBM DB2 Autonomic Technology [10] offers Configuration Advisor, a utility that recommends the best values for the configuration parameters to achieve the goal of a self-configuring system. Similar to Oracle Self-Managing Database [9], Configuration Advisor bases its recommendations on the workload but unlike Oracle Self-Managing Database, IBM's DB2 Autonomic Technology [10] does not have a feature as Automatic Workload Repository and expects the answers about workload characteristics from DBA. On the mission of automating the database design, IBM DB2 Autonomic Technology offers Design Advisor that which includes utilities such as Index Advisor that recommends the best indexes to minimize query execution time. As Oracle's ADDM [9], IBM provides Health Monitor to detect anomalies in critical components of the database manager. IBM's automated framework also contains self-healing features such as Fault Monitor facility to detect faults and recover from failures.

Third, Microsoft SQL Server Self-Tuning Database System [11] provides Index Tuning Wizard tool similar to IBM's Index Advisor [10]. Monitoring the system state is important to enhance self-tuning features and both Oracle and IBM offers monitoring infrastructures in their automated DBMS solutions. For the same reason Self-Tuning Database offers Dynamic Management Views (DMVs) and Continuous Monitoring (SQLCM) framework [12].

In the field of database management and administration, there are a lot of processes that would allow automated procedures to be applied including database installation, configuration and upgrades, backup and recovery, database tuning, monitoring, and migrations. For example, database migrations require a serious evaluation phase between the source and the destination databases. The migration process itself often requires

[7] http://docs.aws.amazon.com/autoscaling/latest/userguide/AutoScalingGroup.html

modifications in backend application that is connected to the database. Even though database systems like relational databases that apply international standards for database definitions and language processing, they develop some different functions, extensions or data types that are not following the standards. To eliminate manual work required for the evaluation of the possible incompatibilities, code and feature set conflicts, database migration evaluation tools are implemented like the Migration Evaluation and Enablement Tool for DB2 or MEET DB2 [13]. In addition, different vendors released a variety of migration tools including Migration Toolkit by [14], Oracle Migration WorkBench [15], and Microsoft SQL Server Migration Assistant [16].

For almost all problems that are faced with manual operations, there are projects that research and implement automated solutions. Some automation projects focused on automating testing procedures is exemplified by [17]. The paper [17] presents a SAT-based approach to automating systematic testing of database management systems, by automatically generating syntactically and semantically valid SQL queries. The approach described in [17], aims to reduce the cost of software development by automating database management testing, which is typically labor intensive and requires complex inputs. Automation of software testing, especially database application testing, studied in other projects in literature, such as MODA project [18]. In the paper [18], researchers presented an automated approach for generating quality tests for database applications. Instead of using the actual database that the application interacts with, they created a mock database from the schema of the actual database and used this mock database to use in test generation.

Dealing with vast amounts of data, modern systems require to be flexible and scalable to handle the workload properly, without affecting the users with long response times as a result of a slowed down application. For this reason, an automated data partitioning strategy that minimizes the cost of expensive data transfers have been researched [19]. The study [19] presents a partitioning advisor that recommends the best partitioning design for an expected workload by advising which tables should be replicated and which ones should be sharded according to specific columns.

Database applications are being updated frequently with the new releases of the application code. Some companies like the social media giant Facebook claim[8] deploying patches daily, in some cases more than once a day[9]. Since the development cycles are getting shorter with the adoption agile methodologies, the logical structure of the underlying database schema is exposed to changes regularly. PRISM project [20] proposes an automated system to (1) predict and evaluate the effects of the schema changes, (2) rewrite queries and applications to operate on the new schema, (3) migrate the database, and (4) invert the migration if necessary.

[8] https://code.facebook.com/posts/495105943907807/ship-early-and-ship-twice-as-often/

[9] https://code.facebook.com/posts/373240506112742/release-engineering-and-push-karma-chuck-rossi/

Configuration management and automation tools such as Ansible, Puppet, Chef or SaltStack allows users to automate their IT systems, hence there are many projects and applications that automate an IT process and also publicly available to other user's access. For example, Ansible Galaxy[10], is the virtual hub where public Ansible roles (prepackaged units of automated tasks) that are written by other Ansible users are stored and shared. Any user can create their own roles, share their roles with other community members, or download the existing roles and reuse them. At the time of thesis being written [April 2017], "PostgreSQL" keyword search on Ansible Galaxy returned 252 roles, "Database" keyword search returned 208 roles, "Cloud" keyword search returned 116 roles, and "Upgrade" keyword search returned 50 roles. As it is seen from the query results, manual processes have been automated by many users with different approaches and tool sets.

The author herself developed a tool[11] to automate PostgreSQL replication in cloud for the PostgreSQL Conference Europe in 2015[12]. The application (or playbook in Ansible terminology) is written by using Ansible and utilizes PostgreSQL and AWS modules. Modules are the building blocks for building Ansible playbooks. In a simple analogy, an Ansible playbook can be thought as recipes while modules are cooking utensils. The playbook automates the whole process from provisioning Amazon VPC and EC2 instances to installing latest PostgreSQL packages on the instances, and finally configuring physical streaming replication with 1 master and 2 standby servers. The playbook also allows adding standby servers to the cluster or removing standby servers from the cluster.

In conclusion, automated software approaches are getting common with the rise of DevOps culture. The movement aims to create a culture where software developers and system administrators collaborate to automate software delivery and system infrastructure to create a rapid, and reliable IT environment. The recent advancements in cloud computing, and the need to manage high volumes of data, force engineers to design more flexible, scalable and reliable systems. Automated software methodologies are emerged to optimize time and human resources while reducing risks related to manual operations. Even though configuration management and IT automation tools are helping developers to build automated solutions for their systems, there are many IT procedures need to be redesign and automate. To the best knowledge of the author, major database version upgrades still require downtime and manual work that is highly dependent on database administrators or system owners. Section 3.2.2 evaluates database upgrade mechanism of different type of databases and emphasizes the need for built-in (native) and automated solutions for upgrades. Therefore, this thesis proposes an automated approach for PostgreSQL clusters in cloud and aims to contribute to the academic and industrial literature of automated database upgrades.

[10] https://galaxy.ansible.com/

[11] Source code of the application https://github.com/gulcin/pgconfeu2015

[12] https://www.postgresql.eu/events/schedule/pgconfeu2015/

3 Technological Background

This section discusses the software methods and technologies that are used for creating a platform which allows automating database upgrades in cloud. Open source technologies are preferred as building blocks of the platform, the importance of which is also discussed below.

3.1 Free and Open Source Software

Before understanding where free and open source software stands in today's world, let us describe what it stands for and what makes a software solution free. First of all, it is "free" as in freedom, not as in "free of charge". In particular, four freedoms define Free Software[13]:

1. The freedom to run the program, for any purpose.
2. The freedom to study how the program works, and adapt it to your needs
3. The freedom to redistribute copies so you can help other users
4. The freedom to improve the program, and release your improvements to the public, so that the whole community benefits.

From the first rule, it is clear that any kind of restriction for the usage of a program makes a software solution non-free. Limited days of free trials, expiring licenses, not being able to run program in specific countries, cities, or limiting the area of usage to limited amount of people, or limited usage for some use-cases like academia-only, or non-commercial. The second rule emphasizes the importance of accessibility of the "human-readable" source code of the program by all means, so that anyone can modify and use it for their own needs and shape it based on the requirement of their systems. The third rule focuses on redistribution policy, if a program does not allow for distributing it to other users, it is also considered as non-free. This rule does not stop software being redistributed with a cost, but only stops being not-redistributed. The fourth and final rule protects the users who can benefit from an improvement of the software, even they cannot program themselves they can still use the program. The programmer or maintainer of the program can charge for this or can release the changes without a charge.

From the explanations of the rules above, one can summarize the philosophy behind free software movement as a trigger for the positive change for the community and people who will use the software for solving their problems. Nowadays, even some countries like in the example of Estonia[14], the USA[15], France[16] and India[17] are adopting open source

[13] https://fsfe.org/about/basics/freesoftware.en.html
[14] https://joinup.ec.europa.eu/community/osor/case/open-source-software-estonia-long-term-policy
[15] https://opensource.com/government/12/9/an-open-source-white-house
[16] https://opensource.com/government/12/11/france-latest-fully-embrace-open-source
[17] https://opensource.com/government/15/8/india-adopts-open-source-policy

standards for their online government systems. This clearly shows the effects of such philosophy on the decisions of the policy-makers at a governmental level which will definitely affect a lot of people’s lives mainly their citizens.

From this perspective, this thesis focuses on using the existing free open source software solutions like Pglogical and PgBouncer, and building new ones for fixing the database upgrade problem in an automated way. An in-depth analysis shows that in terms of code quality, open-source code quality appears to be at least equal and sometimes better than the quality of closed source code implementing the same functionality according to various measures such as maintainability, reliability, extensibility and portability [21].

For creating a tool for database upgrades one should consider the importance of using an open source solution first. When there is a publicly available software solution, it is naturally accessible by users whom they need the solution for solving their problems under their own system architectures. This means a variety of use cases and tests on different production systems from small-sized architectures to very-large-sized architectures under different system loads and requests. Case studies of the open-source solutions that are used in the thesis include PostgreSQL [22] and Ansible [23] back up how these open-source products solve specific business needs, lowered cost of ownership, and reduced deployment time for complex infrastructures and highlight the capabilities of open source solutions.

The Growth of Open Source Software in Organizations Study [24] results show that open source software adoption and usage is on the climb in small to large organizations. When more people started to use open source solutions, they will require more features to make open source tools fitting their own environments better. Since the source code is open and accessible by others, these requests will likely to be handled by other people who also needs that feature or feature set. Hence, the solution will improve over time with the contributions from the other community members. The key is flexibility to modify the source code base on the needs of the IT environment that open source project lives in. A study [25] shows that one of the primary reasons why people prefer using Linux is the ability to modify the source code to meet their needs which proves the importance of being able to modify the code base.

On the other hand, if the solution has a bug or security hole in it, it is more likely to be found, reported and even fixed by the users of the open source solution. The same study [25] shows that people prefer using Linux also because of fast software patches and bug fixes. Open source model encourages users not only for reporting bugs, but actually track them down to their root causes and fix them. For being able to fix the code the users need to understand the code, hence developers review each other’s code; this peer review process helps for detecting the bugs effectively.

Surely open source software has come a long way and it is commonly believed that open source movement is one of the biggest reasons behind the technological improvements of our era by its globally distributed and innovative nature which is shown in many research papers [26] [27].

Conclusively, in the thesis, open source technologies are preferred to work with, in the areas where they prove their capabilities, such as database management, service discovery, and automation. The author believes that free from license fees, usage restrictions, or redistribution issues, open source solutions accelerate the development and deployment processes. In addition, open source solutions allow testing applications that are written on different platforms even with different combinations of open source tools without the risk of vendor locking as in commercial solutions. Furthermore, the freedom and flexibility come with open source solutions make possible to experiment and develop freely, without big-budget limitations of subscription and license fees.

3.2 PostgreSQL and Other Related Database Management Systems

PostgreSQL is chosen as the sample relational database management system [28] over other open source alternatives such as MySQL[18], Firebird[19], SQLite[20] or Apache Derby[21]. The choice of PostgreSQL was based on the fact that it is a popular, open-source relational database management system with a focus on standards compliance and the author of this thesis has had good previous practical experience with it. However, to give a meaningful perspective proprietary products and non-relational database alternatives are mentioned shortly in this section.

The popularity of the database management systems is subject to several studies including The TOPDB Top Database Index project[22]. The project analyzes Google Trends data by looking how often the databases are searched on Google starting from 2005. According to the worldwide study results that are illustrated in Figure 1, Oracle is the most popular database of all and MySQL is the most popular open-source database. PostgreSQL ranks fourth overall and second in open-source databases after MySQL. MongoDB is the most popular NoSQL database and grew the most in the last 5 years (2.1%).

Debian Popularity Contest[23] is another project that aims to find the popularity of database management systems by tracking the database packages installed on Debian[24] (a free and open source operating system) platforms. The project publishes the statistical data of the study participants. Using the published data, the author picked PostgreSQL, MySQL, MongoDB, SQLite and Firebase databases for comparing the popularity of the open-

[18] https://www.mysql.com/
[19] https://firebirdsql.org/
[20] https://www.sqlite.org/
[21] https://db.apache.org/derby/
[22] https://pypl.github.io/DB.html
[23] http://popcon.debian.org/
[24] https://www.debian.org/

source database packages. Figure 2 illustrates how many times a selected database package installed on a Debian platform. The results match with the findings of the TOPDB project (illustrated in Figure 1) that MySQL is the most popular open-source database. However, it dramatically differs on MongoDB, to the result of Debian popularity contest MongoDB is the least favorite amongst the chosen open-source databases. Finally, PostgreSQL keeps its rank as being the fourth database of all.

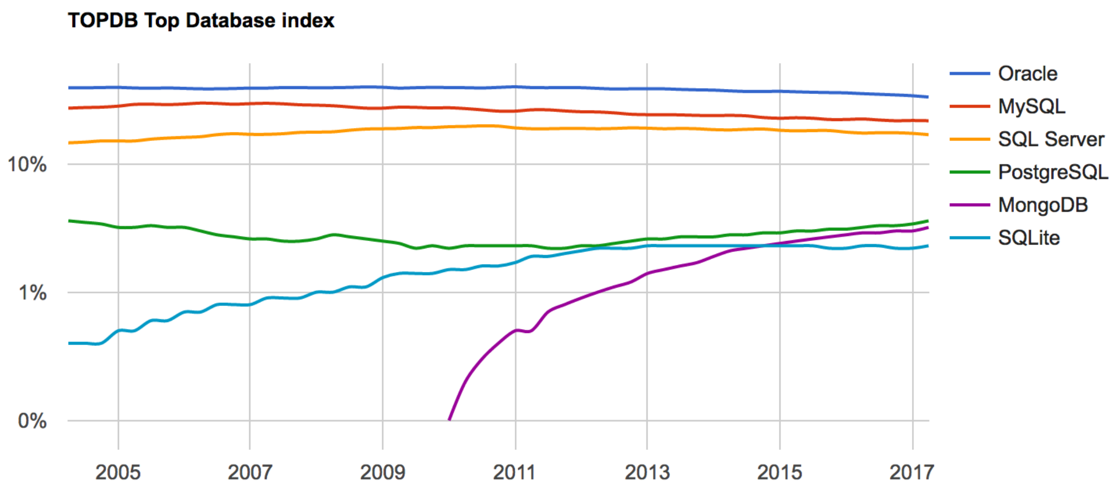

Figure 1. Popularity of top 6 databases based on frequency of Google searches.

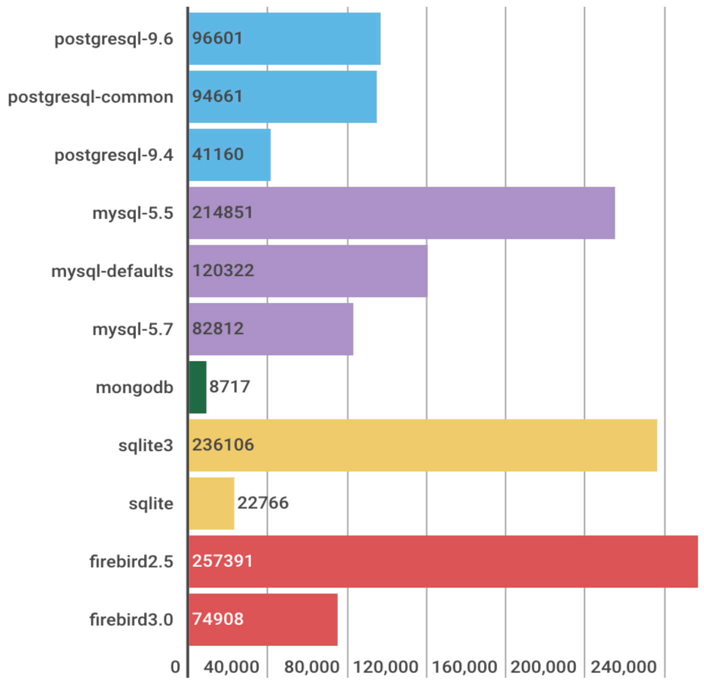

Figure 2. Popularity of Open Source Databases based on Debian package installations[25].

The study [29] conducted by iDatalabs by tracking 50 Database Management System product and technologies found that 594,246 companies using these products. The study

[25] The diagram is created using the statistical data published on http://popcon.debian.org/source/by_inst

contains both open-source and commercial solutions and based on the data they published, the market share of open-source databases is %38. As illustrated in Figure 3, MySQL ranks first among open-source databases and has the biggest market share, PostgreSQL comes next and MongoDB as the third.

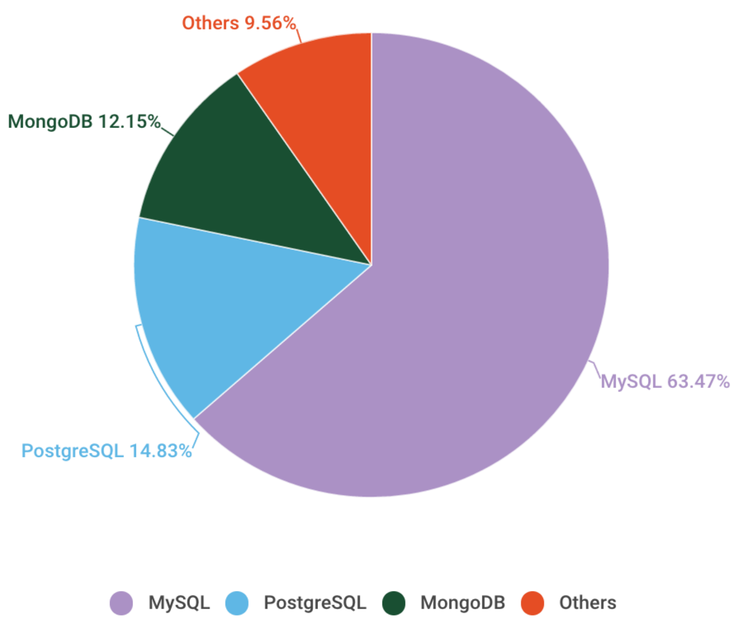

Figure 3. Distribution of Open Source Databases in Business [29].

Before starting to talk about PostgreSQL, it is worth it to mention that PostgreSQL belongs to relational database management system family as its open source alternative MySQL and proprietary counterparts Oracle and MSSQL.

PostgreSQL (in the beginning called Postgres) started as a continuation of Ingres project (1977-1985) by Michael Stonebraker at at the University of California at Berkeley in 1986. The main idea behind Ingres was developing a database system based on Relational Database Management System theory [28] Ingres developed until 1985 and afterwards Postgres (post-Ingres) project started for exploring “object relational” database concept. Postgres was using POSTQUEL query language until two Ph.D. students from Stonebraker’s lab replaced this query language with an extended subset of SQL in 1995, by the name Postgres95[26].

PostgreSQL got its current name in 1996 when a group of open source developers outside of the academia discovered Postgres95 and dedicated themselves to develop the project with many new features and enhancements. PostgreSQL community accepts 1996 as the year of the beginning of PostgreSQL’s new life in the open source world. From that moment, PostgreSQL attracted hundreds of developers all around the world and it has been continuing to evolve with their contributions.

[26] https://www.postgresql.org/about/history/

PostgreSQL is an advanced open source database management system and celebrated its 20th birthday[27] in 2016 with the current major version PostgreSQL 9.6 (PostgreSQL started with the version 6.0 for giving credit to earlier development efforts). Hence it is a proven technology and has an active community, thanks to which it has a fast development progress. Since its earliest versions, PostgreSQL relational database has received various improvements in terms of performance, reliability, availability, and consistency. PostgreSQL runs on all major operating systems, including Linux, Unix and Windows. It is SQL-compliant (SQL:2011) and fully ACID-compliant (atomicity, consistency, isolation, durability). PostgreSQL documentation provides an interactive feature matrix[28].

Database management systems that are based on the relational data model as also called traditional SQL databases because they are using SQL query language like PostgreSQL, MySQL, Oracle and MSSQL. Besides the relational databases there is a rise in popularity of non-relational databases, commonly called as NoSQL databases to show the clear distinction from SQL databases. They are using different querying languages and they mainly do not use traditional table structure for storing data.

Looking through the origins of NoSQL databases, one can easily say that first databases were “NoSQL” as there was no “SQL” yet. Structured Query Language (SQL) came from IBM in the early 1970s and was standardized in 1980s. SQL was built for having a standardized method for accessing and manipulating data in a relational database.

From the 1970’s to 2000’s expectations from the Internet has been changing rapidly, more and more people has access to Internet all around the world. Accessibility to Internet making more business which creates lots of applications hence data has been captured is changing and evolving by years. The need for capturing and storing structured, semi-structured and unstructured data, commonly referred as “Big Data” is getting increased over the years with the help of the cheap storage options. Processing, querying and scaling vast amount of data requires speed, flexible schemas and distributed databases, and NoSQL databases claimed to satisfy these requirements.

Modern NoSQL databases were inspired by the paper Bigtable: A Distributed Storage System for Structured Data [30] from Google. The term NoSQL was first used in 1998 by Carlo Strozzi as a name for his open source relational database that did not offer an SQL interface [31]. The term was reintroduced by Johan Oskarsson of Last.fm in 2009 at an event[29] that he organized to discuss open source distributed, non-relational databases.

For understanding the main difference between traditional relational database management systems and modern NoSQL databases, one should understand ACID

[27] https://thenewstack.io/20-postgres-still-sign-times/
[28] https://www.postgresql.org/about/featurematrix/
[29] http://blog.sym-link.com/2009/05/12/nosql_2009.html

(Atomicity, Consistency, Isolation, and Durability) and BASE (Basically Available, Soft state and Eventual consistency) terms. Relational database management systems are ACID-compliant and NoSQL databases are considered as following BASE principles. Most of NoSQL systems do not attempt to provide ACID guarantees, contrary to the prevailing practice among relational database systems.

Atomicity ensures that all commands in a transaction are either succeeds or fails, there is no in-between state preserved. Consistency enforces that all committed data is consistent according to the rules and system-defined constraints which are defined in database. Isolation provides control over what other clients can see; there are different transaction isolation levels which are supported at different levels in various database systems. Durability ensures that once data is written it will be always there regardless of failures or crashes.

Most NoSQL databases lack true ACID transactions and offer BASE concept which is diametrically opposed to ACID concept. Most distinguishable part of BASE is that it offers “eventual consistency” by which it declares that system will be consistent over time (typically within milliseconds). This might result in reading data that is not accurate, because the data might not be updated by the recent changes immediately. In other terms, ACID is pessimistic and forces consistency at the end of every operation and BASE is optimistic and accepts that the database consistency will be a state of flux [32].
CAP Theorem [33] states that web services cannot ensure Consistency (has different meaning than consistency in ACID) , Availability and Partition Tolerance at the same time. BASE applications have their focus on trading consistency for availability. As a result of this tradeoff, “Eventual consistency” allows BASE applications to achieve higher levels of scalability that cannot easily obtained with ACID.

“Soft state” comes as a result of “Eventual consistency” model. The state of the system may change over time even without client activity (without input) due to changes going on to make the system consistent over time, thus the state of the system is always “soft”. One can easily conclude that, soft state logic abandon the consistency requirements of the ACID model pretty much completely.

“Basically Available” mentality of BASE is achieved through supporting partial failures without total system failure which lead to a higher perceived availability of the system. CAP Theorem’s “Partition Tolerance” corresponds to “always” available “Basically Available” state of BASE applications. Operations will complete, even if individual components are unavailable. Web applications need to make the decision between consistency and availability if they have horizontal scaling strategy based on data partitioning. The term horizontal scaling comes from how the increase in hardware capacity is achieved. Traditional database architectures are designed to run well on a single machine, and the simplest way to handle larger volumes of operations is to upgrade the machine with a faster processor or more memory. That approach to increasing speed is known as vertical scaling. More recent data processing systems, such as Hadoop and

Cassandra, are designed to run on clusters of comparatively low-specification servers, and so the easiest way to handle more data is to add more of those machines to the cluster. This horizontal scaling approach tends to be cheaper as the number of operations and the size of the data increases, and the very largest data processing pipelines are all built on a horizontal model. There is a cost to this approach, though. Writing distributed data handling code is tricky and involves tradeoffs between speed, scalability, fault tolerance, and traditional database goals like atomicity and consistency

NoSQL databases are differing from each other in many ways hence there is a need to categorize them like document store (document-oriented), key value store, wide column store (column families), graph store etc. Apart from the points that they try to achieve in common, such as being non-relational, distributed and horizontally scalable they are mainly optimized in what kind of data they store and how they store it (data model).

For storing data in a traditional database, the user first defines column types and column names and creates the table where data will be stored. Then data is inserted as rows of values into the columns as a cell of each row. This approach does not allow to have additional values that were not specified when the table is created, and also every value must be present, even if it is a NULL value. When NoSQL databases claim being “schemaless” they refer being “schema-on-read” instead of “schema-on-write” which is how traditional data storing works as explained above: define the schema first, write the data into it, read the data which comes back in the schema that is predefined.

NoSQL databases have been oriented towards the schema-on-read approach. Document stores such as MongoDB, RethinkDB and CouchDB allows users to enter each record as a series of names associated with values. Even though the values have some specific format (i.e. JSON, BSON, XML), users do not need to specify what names will be in each table using schema. Parts of the value can be manipulated as long as the application layer does not rely on the values that were removed. This brings more flexibility into document-oriented approach comparing to relational databases where “schema-on-write” is applied, on the other hand “schema-on-read” requires that application layer needs to be aware of all versions of the schema [34].

Storing key-value pairs is another category in NoSQL databases such as Redis, Memcached and RocksDB. In document stores, database is aware of the structure and contents of the individual documents, but in key-value stores, value is opaque to the database. Unlike document stores which allow users access to data using part of the value, key-value stores allows access to the data only through the unique key. The operations can be listed as getting the data associated with a particular key, storing some data against a key, and deleting a key and its data. This simplicity makes scaling a lot easier, but application layer has to handle building any complex operations which normally would be handled in relational databases.

Another way of storing non-relational data is wide column store which has applications in Cassandra and Hadoop. These databases store records (or rows) which contain arbitrary amount of columns. In other words every record can have different columns. Wide column stores are originated from the Google's BigTable paper [30]. There are other special type of NoSQL databases that are optimized for storing different types of data such as graphs databases, but those are out of the focus of this thesis.

There is a new term worth to be mentioned is called NewSQL, which are influenced by the designs proposed in Michael Stonebraker's "The End of an Architectural Era (It's Time for a Complete Rewrite)" paper [35]. They represent a new wave of database systems that retain many features of the relational model but also enhance or modify the fundamental principles of the underlying technology in significant ways. Unlike NoSQL databases, NewSQL databases such as Vertica, VoltDB, NuoDB and MemSQL employ consistency models of the traditional RDBMS, ACID transactions and multi-version concurrency control (MVCC). They also use SQL language unlike NoSQL databases, in a way with NewSQL movement SQL has found its way back into the world of non-relational databases.

In conclusion, NoSQL databases remove ACID in database layer by pushing consistency problems into the application layer where they are not any easier to solve. They also sacrifice SQL, which makes querying more complex comparing to SQL solutions. NewSQL systems[30] benefit high-level query capabilities of SQL, and offer high performance and scalability while keeping ACID transactions.

Conclusively, it is very important to understand commonalities and differences that database management systems have. Either relational or non-relational, open source or commercial, database management systems share the same value proposition of storing data and making the data retrieval convenient as possible for the users by different implementations.

Upgrades are essential for all database management systems and for being able to create a platform that automates database upgrades, different database design approaches evaluated and discussed briefly in this section. The author believes that the success of the platform applicability to the other database management systems is highly dependent on the research about different database methodologies prior to the platform design.

3.2.1 Overview of PostgreSQL

In this thesis, PostgreSQL is chosen to apply automated upgrade platform approach. There are several reasons behind this choice. The author has experience with PostgreSQL in production systems for the last 6 years. PostgreSQL uses rich and standardized SQL query language, which allows users to do most of the operations over the data inside the

[30]https://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext

database. PostgreSQL offers many options regarding to the storage patterns; including relational (normalized or not) data, document-based data (i.e. JSON, XML), key-value pairs and many more. PostgreSQL is fully ACID-compliant and more mature comparing to non-relational counterparts. Over the years, PostgreSQL has picked features from non-relational databases and integrated them into its core, or provided some functionalities through extensions.

PostgreSQL ranked in fourth place behind Oracle, MySQL and Microsoft SQL Server in DB-Engines' ranking[31] of the most popular databases for February 2017. This result shows that, PostgreSQL is more popular than the most popular NoSQL database on the list which is MongoDB (ranked in fifth place). Unlike commercial databases that are proprietary, expensive and require vendor lock-in, PostgreSQL is open source and enterprises have been moving to open source options which explains the popularity of PostgreSQL.

PostgreSQL has native JSON support, which allows users to start unstructured, but then also convert it to structured data over time if the requirements change. As the application matures, users have the flexibility to keep some of their data remain unstructured and some of it strictly structured. For another way of keeping different data stores, PostgreSQL has support for Foreign Data Wrappers (FDW) which allow PostgreSQL Server to access different data stores, ranging from other SQL databases (i.e. MySQL, Oracle, MSSQL Server) through to CSV or JSON files. In other words, data wrappers allow users to connect a remote system within PostgreSQL, then read and write data from other databases and use it as if it were inside the PostgreSQL database. They allow PostgreSQL queries to include structured or unstructured data, from multiple sources, such as NoSQL databases (i.e. MongoDB, Cassandra, Redis) or even other PostgreSQL databases.

3.2.2 Upgrades in Database Management Systems

In this section, upgrades in different database management systems compared shortly to give a better understanding why automated upgrade approaches are still a valid need in database world. For making a reasonable comparison with PostgreSQL popular alternatives are chosen. MySQL is chosen for being an open-source and relational; MongoDB is chosen for being an open-source and non-relational; Oracle is chosen for being a commercial and relational database management system.
In PostgreSQL, for major release upgrades, there are three possible paths that can be taken. First method is upgrade by restoring from a logical dump, second one is physical upgrade and the last method is online upgrade.

[31] https://db-engines.com/en/

First and the most tested method requires a logical dump using pg_dump[32] from the old version, and pg_restore[33] on a clean cluster created with the newly installed version. This method gives flexibility to upgrade from very old versions (goes back to version 7.0's, which were released by 2000) to one of the recent releases. Many other database systems (MySQL, MongoDB, Oracle etc.) and upgrade methods lack the ability to jump from very old releases to new releases, most of the time supported upgrades are limited between the latest version and the previous major version. For being able to start running pg_dump, write connections to database have to be stopped, and especially on large databases this will create a huge total downtime. Another disadvantage to this method is that it requires double disk space, or the removal of the old cluster before restoring.

Second method is called "in-place" upgrades because the process is done over the same server and preferably on the same data directory. It is also categorized by physical upgrade because this method does not require logical dump/restore processes. A clear advantage to this, since there is no logical dump required, there is no extra space needed for another copy of the cluster. In PostgreSQL, this method is provided by pg_upgrade[34] tool and pg_upgrade supports upgrades from version 8.4 and later to the current major release of PostgreSQL. Major releases come with new features that often change the layout of the system tables, but internal data storage format rarely changes in PostgreSQL. PostgreSQL community tries to avoid changing internal data storage and this is an advantage that pg_upgrade uses as a benefit. The pg_upgrade tool performs upgrades by creating new system tables and reusing the old user data files. As a result, downtime is much lower compared to using pg_dump. Even though it is considered faster than pg_dump method, it has some disadvantages. Once the new version of PostgreSQL is started, there is no way to go back to the old version. Cluster will work only with the new version from there on. (There is a way to run pg_upgrade to make it generate a second copy on disk of the cluster but then there is almost no advantage of using this method over pg_dump). It will perform poorly in clusters with many databases, or databases with many thousand objects.

Business continuity requires a better method for database upgrades ideally achieving zero-downtime upgrades which are not the case with pg_dump/pg_restore and pg_upgrade methods. The findings of a recent survey [3] that was conducted in 2015, with more than 200 enterprises involved, shows that enterprises cannot afford to maintain the status quo when it comes to database availability. More than 70 percent of enterprises report delaying database upgrades because of concerns over downtime in their critical applications. Survey results show that having the most critical applications be offline for 20 minutes to three hours, more than once a month is not acceptable to any enterprise today. It is clear that modern database management systems need to improve their upgrade mechanism to achieve better results with minimal downtime.

[32] https://www.postgresql.org/docs/current/static/app-pgdump.html
[33] https://www.postgresql.org/docs/current/static/app-pgrestore.html
[34] https://www.postgresql.org/docs/9.6/static/pgupgrade.html

The third and last method for PostgreSQL upgrades is called online upgrade. This type of upgrade has been available since the first trigger-based replication solutions (i.e. Slony-I, Londiste[35]) were developed even before PostgreSQL had a built-in replication support. (Replication methods and how they are used for database upgrades is covered in details in Section 3.2.3.) Upgrading PostgreSQL by using trigger-based replication solutions requires several steps to follow. First, both versions (the existing one that is going to be upgraded and the new one that is going to be upgraded to) should be installed, this gives a flexibility to have them working in parallel without requiring a downtime. Then, on the source node (the current existing version) an initial copy should be created, and the changes should be replicated to the other node. The changes from the moment when the copy operation started, eventually will be replicated to the new server. That is why logical replication should be kept until the replication lag is close to zero. The final step is only repointing the connection info from the application server to connect to the new server.

Online upgrade method is very convincing in the sense that making zero downtime upgrades possible. Another advantage is that trigger-based replication solutions allow upgrades regardless of which version is running on the nodes, the changes are copied using triggers (supported SQL commands). They allow online testing of the new cluster by read-only queries, so that if there is a problem there is a way to cancel the upgrade operation without damaging the old cluster version. Even though this method has many advantages comparing to the previous two methods, there are some major disadvantages worth mentioning. Like pg_dump/pg_restore, it needs double storage space, as it has to store the second copy of the data. All the changes are captured by using triggers, and written into queue tables. This procedure doubles the write operations, doubles log files, and slows down the system since all the changes has to be written twice; resulting more disk I/O and load on the source server.

In the light of comparison of the existing PostgreSQL upgrade methods, the author chose to use pglogical tool for this thesis. Although the method that is chosen is categorized as logical replication, it has different implementation than trigger-based solutions. The reasons for the choice of pglogical is explained in details in Section 3.2.3.

Second database is chosen for checking upgrade procedures is MySQL, and it supports two methods[36] for database upgrades. First method is called "in-place upgrade" and similar to the current method for PostgreSQL minor version upgrades and running pg_upgrade on top of that. Steps involve shutting down the old MySQL version, replacing the old MySQL binaries or packages with the new ones, restarting MySQL on the existing data directory and running mysql_upgrade[37] tool which checks all tables in all databases for incompatibilities with the new version of MySQL server, if problems are found, attempts a table repair.

[35] https://github.com/pgq/skytools-legacy

[36] https://dev.mysql.com/doc/refman/8.0/en/upgrading.html#upgrade-methods

[37] https://dev.mysql.com/doc/refman/8.0/en/mysql-upgrade.html

Second method for MySQL upgrades is called “logical upgrade”, which is similar to PostgreSQL major upgrades via pg_dump and pg_restore. Data from the old MySQL version is dumped with mysqldump[38] tool, the new MySQL version is installed, and dump file is restored into the new MySQL version and finally mysql_upgrade tool is run. MySQL documentation recommends executing mysql_upgrade for any kind of upgrade operation. As discussed above, MySQL does not have a built-in automated upgrade solution.

MongoDB is chosen as a representative of NoSQL databases, and MongoDB upgrades are also not automated and have some requirements over versions. For instance, to upgrade an existing MongoDB deployment to version 3.2, system must be running on a 3.0-series release; or to upgrade to version 3.4, system must be running on a 3.2-series release. Users also need to check compatibility changes document[39] for the version that they will upgrade to. Ensuring the applications and the deployments are compatible with the new MongoDB version is the responsibility of the users (system owners, system administrators, database administrators). Resolving the incompatibilities before upgrades are crucial for the upgrade to succeed.

Upgrading MongoDB differs whether it is a standalone MongoDB instance, a replica set (MongoDB defines group of instances with the same data set as replica sets, similar concept called as a replication cluster in PostgreSQL or MySQL), or a sharded cluster (MongoDB calls a cluster as sharded cluster if each shard contains a subset of the sharded data, which can be deployed as a replica set. Sharding is the way for scaling data horizontally within the cluster. PostgreSQL does not have a built-in sharding solution in core yet, but there are solutions as Postgres-XL[40] which implements scaling for PostgreSQL.) Upgrading MongoDB requires manual operation for the three of the scenarios that are listed above. There is an option to upgrade a standalone MongoDB instance via package managers (i.e. apt, yum) if only MongoDB is installed via packages (i.e. deb, rpm) on Linux distributions (i.e. RedHat, Debian, Ubuntu). Nevertheless, this still means downtime and requires shutting down the mongod[41] (is the primary daemon process for the MongoDB system) instance, replacing the existing binary with the new mongod binary and restart mongod. To conclude, MongoDB upgrades are also works in a manual fashion.

Oracle is chosen as a representative of commercial databases. The terminology that has been used in Oracle white papers are full of different product names and solutions (i.e. Database Upgrade Assistant (DBUA), Oracle Active Data Guard Far Sync, Oracle Multitenant, Oracle Fusion Middleware, Oracle WebLogic, Oracle Universal Installer (OUI), Repository Creation Utility (RCU), Reconfiguration Wizard) and this results in an

[38] https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html

[39] https://docs.mongodb.com/v3.2/release-notes/3.2-compatibility/

[40] http://www.postgres-xl.org/

[41] https://docs.mongodb.com/manual/reference/program/mongod/#bin.mongod

unnecessary complexity over a key system operation as database upgrades. Documentation itself is very unstructured and backwards-incompatible for an open-source background person as thesis author. Many of the methods are only applicable for a subset of main Oracle versions; depending on which operating systems and the versions that the source and destination are running on, what are the hardware platform and the size of databases etc. There is an “in place” upgrade method for upgrading an Oracle database by using Database Upgrade Assistant (DBUA) or command-line upgrade scripts. These are two variants of the same method, DBUA offers GUI that executes the same command-line scripts. As referred as “in place” upgrade, it does not require creating a copy of the existing database. There are other methods that are simply depending on dump/restore principles and applicable in different scenarios. These methods require considerable amount of downtime and they do not claim to meet minimal downtime requirements. However, there are strategies to minimize downtime for Oracle upgrades by using the solutions like Oracle Data Guard and Oracle GoldenGate, which are covered in Section 3.2.3.

There are services offered by companies to upgrade databases. For instance, Intelligent Upgrade Robot[42] is a paid service for upgrading Oracle databases.

For all the chosen sample databases, version upgrades require downtime and manual work to a certain degree. In conclusion, built-in automated upgrades are not a standard procedure for popular database management systems independent of being open-source or commercial; relational or non-relational.

3.2.3 Replication in Database Management Systems

Upgrade methods were discussed in Section 3.2.2 and this section covers how database replication methods are used in minimizing downtime in database upgrades. Database replication is the general term for describing the technology of maintaining a copy of a set of data on a remote system. In this thesis, the focus is on logical replication methods, hence the mechanism allows online database upgrades in addition to their main functionality as being database replication.

In PostgreSQL, each change made to the database (inserting a row into a table, creating an index etc.) is recorded first in binary files called WAL (Write-ahead Log), hence the name "write-ahead" log, as a synonym of "transaction log". Should PostgreSQL crash, the WAL will be replayed, which returns the database to the point of the last committed transaction. Write-ahead logging mechanism is the main fault tolerance system for PostgreSQL which ensures the durability of any database changes.

PostgreSQL has Physical Streaming Replication in-core as state of the art of replication in version 9.6 and the replication system is also based on WAL mechanism. Clients execute queries on the master node, the changes are written to a WAL file and copied

[42] http://infuse.it/solutions/intelligent-upgrade-robot/

over network to WAL on the standby node. The recovery process on the standby then reads the changes from WAL and applies them to the data files just like during recovery. To put it briefly, streaming a series of physical changes from one node to another is called Physical Streaming Replication which is illustrated in Figure 4.

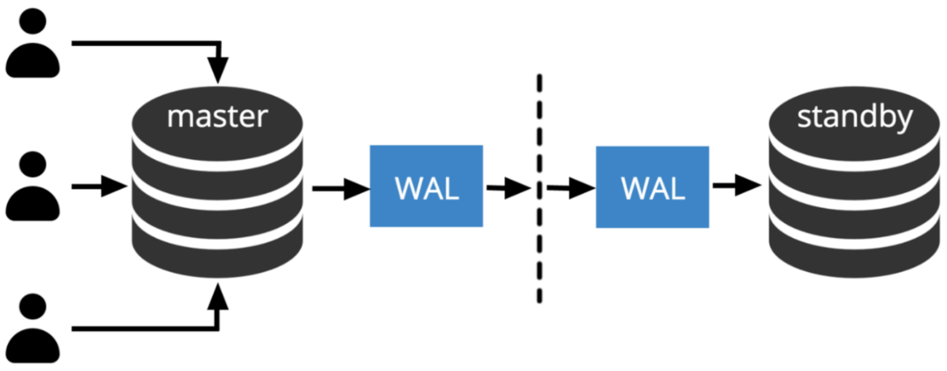

Figure 4. Physical Streaming Replication.

Logical replication allows streaming logical data changes between two nodes. Unlike physical replication which captures changes to the raw data on disk, the logical replication captures the logical changes to the individual records in database and replicates those. The logical records work across major releases, hence logical replication can be used to upgrade from one release to another. There are two basic approaches to logical replication: the trigger-based and the changeset extraction (called logical decoding in PostgreSQL). Using trigger-based replication solutions for PostgreSQL upgrades, is mentioned in Section 3.2.23.2.2 Upgrades in Database Management Systems. Logical decoding and the advantages of using it for minimal downtime database upgrades is the focal point of this section.

In 2014, PostgreSQL has introduced Logical Decoding in version 9.4 and this feature opened the door for a whole new world of possibilities. Logical decoding is a mechanism that extracts information from WAL files into logical changes (INSERT/UPDATE/DELETE). Since the data from WAL mechanism is used by decoding the transaction logs, there is no write amplification as in the case of trigger-based replication solutions, hence this method performs better[43]. Logical replication is built on top of logical decoding. At the time of this thesis being written, logical replication is only available as extensions. Pglogical extension is used for implementing Automated PostgreSQL Upgrades Platform in this thesis. Pglogical serves as base for built-in logical replication solution in the next major version of PostgreSQL (version 10) and this is one of the main reasons why Pglogical has been chosen to implement thesis application.

Logical replication is also used for upgrading Oracle databases. Oracle calls this upgrade method as “transient logical database rolling upgrade process” which is performed by

[43] https://blog.2ndquadrant.com/on-pglogical-performance/

using Oracle Data Guard[44]. Oracle Data Guard can be used in similar fashion as Pglogical to replicate between major versions of an Oracle database. However, due to restricted licensing users are required to either always reuse the existing servers or to buy new licenses which will not be used after the upgrade has finished which limits the usefulness of this approach. Moreover, the upgrade operation requires manual operations in some phases, different products need to be used, pre-upgrade and post-upgrade tasks and scripts need to be executed, and it requires downtime in certain phases. Even though this method provides a step towards near-zero downtime, there is still room for improvement.

3.3 Connection Management using PgBouncer

When a database upgrade operation is completed, any application that is using the database server should point to the upgraded new cluster. Achieving zero downtime upgrades require automated service discovery to avoid shutting down the application, pointing to the new cluster and starting the application again. In the thesis, PgBouncer tool is used to ease the transition from old server to the upgraded new server.

PgBouncer is an open-source PostgreSQL connection pooler originally developed at Skype. PgBouncer acts as Postgres server, so any target application can be connected to PgBouncer as if it were a PostgreSQL server. In the thesis, this tool is utilized to achieve a graceful upgrade operation without dropping active database connections.

Upgrading a database cluster needs an external tooling or configuration setup to allow applications to point the new cluster and this need is not only PostgreSQL’s concern. As mentioned in Section 3.2.3, using Oracle Data Guard to perform transient logical database upgrades require a brownout, to change the roles of the databases. As suggested in Oracle documentation, when the goal is absolute zero downtime then Oracle GoldenGate tool must be used in place of the Data Guard database rolling upgrade process [36].

3.4 Cloud Computing

Cloud computing platforms (AWS in specific) are chosen to implement Automated Database Cluster Upgrade Platform for this thesis. There are several reasons behind why cloud computing platforms are chosen instead of traditional physical servers that are briefly discussed in this section.
One can argue that many of the concepts of cloud computing have been around for decades, which is valid to some extent. The road to the cloud computing is paved with techniques that are learnt from mainframe computing era through to Internet era.

In the very first concept of cloud computing model is that instead of saving the data of the applications from a local computer or a server housed in an on-premise data center,

[44] http://docs.oracle.com/database/121/SBYDB/concepts.htm#SBYDB00010

there is the ability to run the applications without owning any hardware or a server with pay-as-you-go billing model, at a massive scale. Even though cloud term sounds like a magical place to save the data, the truth is that the application data is actually served on larger, more powerful systems in a remote place that the resources (processing, networking, and storage) are mostly shared and distributed among many users of the cloud service providers.

Centralized computing of the mainframe era was built on "time sharing" concept which can be explained as sharing of a large computing resource by different customers. The concept of sharing resources of cloud computing is very similar to mainframe computing in this regard. Main motive behind both cloud computing resource sharing and mainframe computing time sharing is reducing the cost of computing capability. Cloud computing allows users to use cloud services, provision and deploy instances quickly without owning any hardware. Same logic was applying to mainframe computing, they were extremely expensive machines, also required entire rooms or buildings to be placed. In this sense, cloud computing technology is giving back the central control of mainframe computing by also offering broad network access over the Internet, and the ability to start using the instances in a short amount of time and stop using when the need is over, and paying only the amount that has been used in terms of resources and services.

As with every technology, there were some caveats in mainframe computing. Centralized computing was not flexible enough for the companies that wanted more control over the systems, this lead to "mini-mainframes"[45] came to life. With this technology, companies could own their own mainframes instead of renting time on a system used by other companies, hence control their resources better. The technology got more advanced over time and Personal Computer (PC) revolution made computers a lot cheaper than their ancestor mainframes. The need of the information transfer between PCs triggered networked PCs, so that computers could share the information between them over cables.

Cloud computing technologies made the computing resources available from anywhere in the world, anytime when companies or individuals want to access or stop using the resources. Before the cloud technologies, planning a migration project or starting to a new software project was involving long time spans of hardware resource planning, research about licensing and upgrade methodologies, feasibility analysis, supported platforms and for how long that support would be available, what would be done with outdated instances and hardware that was replaced and many more projections over how hardware and software would work out together for the expected lifetime of the project. Cloud technologies simplified and automated these services and made them available as highly abstracted on-demand services with clear and affordable price points, that can be turned on and off anytime [37].

[45] https://techtalk.gfi.com/from-mainframe-to-cloud-its-technology-deja-vu-all-over-again/

Modern projects do not need to estimate and plan resources years in-front without enough data available for projecting over the future of the application. Managing and scaling the resources by the usage expands, or terminating and shifting the resources which are idle or less busy to the areas that require more resource is a matter of clicks. The elasticity of increasing or decreasing the size of an instance, adding hundreds of them together, scaling the load by adding tens or hundreds of servers concurrently, that are globally distributed is what makes cloud better than the initial centralized computing methods. Author, herself is working from Tallinn for a company based in London, having colleagues from 23 countries with only a working laptop and Internet connection. She can provision and deploy instances in Tokyo, California or Mumbai from the web console of a cloud provider, manage the applications of the company's hundreds of customers from Australia, Dubai, or France with the applications that run in cloud and accessible to all employees and customers.

Cloud computing considered as an evolving paradigm [38], and as in with every new technology standards and best practices are missing amongst many cloud providers; by time and expansion of the usage, cloud computing best practices will be clearer and technology will be more standardized. Even though for software solutions, the author preferred open source solutions, Amazon Web Services (AWS)[46] is chosen for implementing Automated Database Cluster Upgrade Platform in Cloud. Surely there are open source cloud computing platforms for example OpenNebula[47], Eucalyptus[48], OpenStack[49] but the author has hands-on experience with AWS through her work experience and it was chosen for reducing the cost of learning a new technology and risks are related to the less known platforms.

Finally, it may be concluded that the accessibility of cloud computing resources, pay-as-you-go flexibility, ease of experimenting with several sizes of instances, and scalable nature of cloud computing enabled a fast and affordable solution to implement Automated Database Upgrade Platform for the thesis. Otherwise, setting up physical servers, configuring them properly and efficiently would require more time, and cost of ownership would be much higher. As can be seen in Chapter 6, one of the case studies require adding 20 new servers to the cluster and removing 20 old ones during the test. Without the cloud infrastructure, setting up 20 new instances and throwing out 20 old ones would be impractical at the least. Fortunately, provisioning of the cloud instances is not difficult as setting up traditional servers manually. In fact, setting up 20 cloud instances is not any different than setting up one cloud instance time-wise and operation-wise. And the cost of additional instances is only temporary.

[46] https://aws.amazon.com/
[47] https://opennebula.org/about/project/
[48] https://github.com/eucalyptus/eucalyptus
[49] https://www.openstack.org/

3.4.1 Service Models

National Institute of Standards and Technology (NIST) defines three service models for Cloud Computing [38]: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Each model gives a different level of control of the deployed applications in cloud, by automating remaining tasks and providing them as a service. Before cloud systems, in a traditional on-premises data center, IT teams were responsible of building and managing all infrastructure, hence they had full control over their systems. Cloud service models provide a level of abstraction for the cloud applications, and IT teams can focus on developing their applications rather than managing network, storage, database, and operating system layers, or they can manage these IT elements at a scale and in a more automated way.

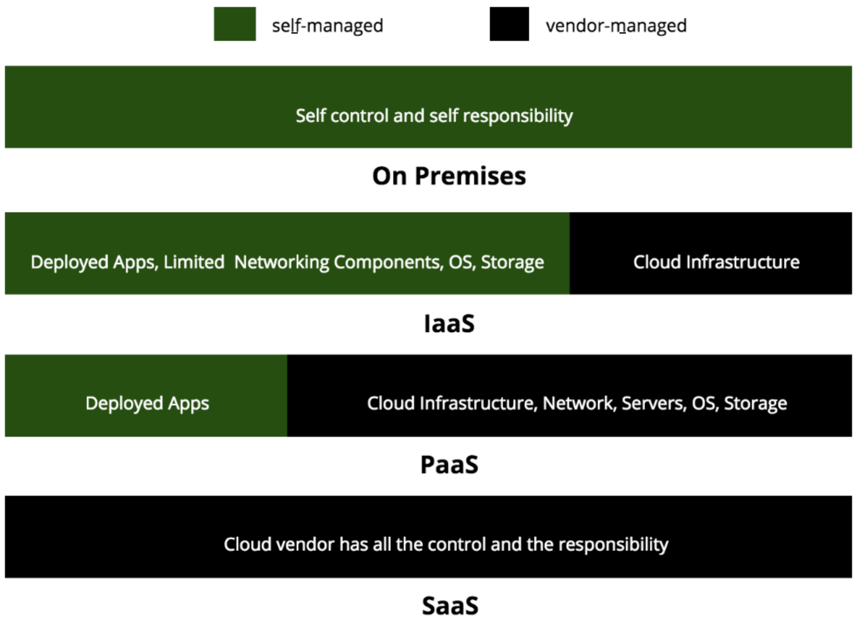
self-managed
vendor-managed
Self control and self responsibility
On Premises
Deployed Apps, Limited Networking Components, OS, Storage
Cloud Infrastructure
IaaS
Deployed Apps
Cloud Infrastructure, Network, Servers, OS, Storage
PaaS
Cloud vendor has all the control and the responsibility
SaaS

Figure 5. Cloud Service Models.

The definitions of service models from NIST, made the illustration easier for the author. As it can be seen in Figure 5, the difference between IaaS, PaaS, SaaS and On-Premises is mainly depending on who is in charge of the application and infrastructure beneath either cloud provider or the company who uses the cloud services. In the case of On-Premises there would be no cloud services available and companies were managing all the systems and components related with their applications themselves.

IaaS is the model where companies/users have the most autonomy over their system, cloud provider manages and controls the underlying cloud infrastructure, as service is being infrastructure, and company who uses the cloud service gets the rights to deploy and run their software. In this model, provisioning process, storage, networks, operating systems and applications are responsibility of the paying customer (either company or individual) and cloud provider neither controls nor manages these computing resources, only provides the cloud infrastructure that allows consumers to manage their resources, build and deploy their applications.

From the perspective of the cloud services consumer, in the decreasing order of control of the system, PaaS comes second after IaaS. Cloud provider serves the platform by offering cloud infrastructure including network, servers, operating systems and storage. The consumer of the service has the control over the deployed applications and possibly configuration settings related to their application and hosting environment. The cloud platform which is provided by the cloud vendor, supports some programming languages, libraries, services and tools. Consumers of the platform service, could deploy their own or acquired applications onto the cloud infrastructure by using these offered tools, libraries and programming languages that are supported by the cloud vendor. Even though, this capability does not necessarily prevent the use of alternative set of tools and the software stacks, it would not be wrong to say that there is a certain limitation for the application developers while using PaaS coming through cloud vendor. The ease of using a platform to develop cloud applications without dealing with the underlying infrastructure, comes with a subtle price: developers almost have no control over low level software controls like memory allocation, caching, number of threads etc.

The last cloud model is SaaS, which can be summarized as having a software solution runs in cloud, and paying a subscription fee to use the application over the Internet through a web interface. Consumers do not have any control over the underlying cloud infrastructure or even in the application itself; only have rights to use the application and might configure some user-specific application settings. SaaS provider has full responsibility over the software application and its performance, security, scalability, privacy and IT elements running beneath it, consumer only has to pay a price for the service usage.

In conclusion, each service model offers a different level of flexibility to the consumers by automating underlying IT tasks and utilizing the cloud services. Consumers should consider the requirements of their IT environments and make the decision accordingly by understanding capabilities and restrictions of each model.

In this thesis, IaaS model is used for provisioning the cloud instances and deploying Automated Database Upgrade Platform application. The IaaS model provides enough control which is needed to install and use all the required components that are not available in PaaS and SaaS situations, like the Pglogical extension or SSH access for Ansible, while still providing more flexibility than on-premise solution thanks to a dynamic environment with pay as you go price model.

3.4.2 Deployment Models

NIST defines four cloud deployment models: Private, Public, Community and Hybrid Cloud. Cloud deployment models differ from each other by factors like scalability, flexibility, cost, security and customizability, hence each model fits best for different system and business requirements.

One can use a simple analogy to describe different cloud models. If each cloud model was a living plan, a public cloud would be an apartment where multiple tenants living at the same apartment within different flats but they also have common places and shared utilities. Following the analogy, a private cloud would be a family owned house with different rooms, which are mainly used by different family members, but also have common areas like living room and kitchen. Using the same analogy, a community cloud would be like a hippie camp where only members of a peaceful community with similar views, and concerns can join and use the camp area. Lastly, a hybrid cloud would fit into a mix of other living plans like the one that is offered by the popular renting application Airbnb[50], where people rent the flats or houses that they own to different tenants for a short period of time. Flat or house owners can still use some of the rooms privately for their usage, and rent other parts of their houses and that would be fitting to hybrid cloud model in the analogy.

Evaluating the requirements of Automated Database Cluster Upgrade Platform to determine which type of cloud is the best fit in respect of the thesis application, Public Cloud model is chosen for the implementation. The primary benefits of the public cloud include the speed and pay-per-use policy. Mainly, public clouds are ideal for any kind of project which has a need for fast deployment include the thesis period with a tight deadline of one semester. In addition, public clouds are based on shared physical hardware (with other tenants) which is owned and operated by the public cloud provider and usually cheaper than the dedicated private clouds. Lastly, public cloud model gives a higher flexibility to scale (add or drop capacity) within the capacity of the cloud provider (i.e. AWS, Google Cloud, Microsoft Azure) which is likely to be higher than a sole private cloud owner’s computing capacity.

Although Private Cloud model has its own advantages over Public Cloud model, none of the benefits were required for the thesis implementation. For example, if the thesis application would have required more security and privacy, then having a dedicated private hosted environment would be a better choice which is offered by the private cloud. Another reason for choosing private cloud would be the requirement of the need for very specific hardware (or configuration) to solve intensive computational problems. In this case, the public cloud would not be an option, simply because the edge-case computational needs usually not offered by the cloud vendor for the general public use.

A private cloud could be thought of owning a car, then you would surely know how much it will cost for you to fill the fuel tank. Similarly, a private cloud comes with a predictable cost and it is better if the application will need this resource with the everyday load. Using the same analogy, if you have a family and need to use the car for dropping the kids the

[50] https://www.airbnb.com/

school, going to shopping, regular hospital visits, going to work, picking up the kids, visiting friends and family, then you know that the car is helpful and owning this car gives you more control for organizing your life, and better management visibility since you drive the car and you pay for the predictable cost.

A public cloud could mean not owning the private car in the analogy, but instead using public transport like bus or tram for everyday commute and ordering a taxi when you are late to a business meeting, or the weather is inconvenient for using the public transport, or the bus is full and you are too tired waiting until the next bus comes. As the analogy reveals, a public cloud is flexible and you do not need to pay as much for filling the fuel tank of a car every month, which is more expensive than having a monthly ticket for the bus. When you need to use faster and more expensive service from time to time, as a taxi in the analogy, you are free to expand your computing resources for an amount of time, like a spike in your system load that would require immediate action (scaling out the cluster or adding more memory, disk, CPU etc.).

When you own the car, you are free to customize the features of your car, but you cannot expect to add a speed wheel to a bus. As the owner of the private car, you can modify and customize your private car as you like, but there is always a limit for that. You cannot exceed the physical limits of the car. On the other side of the story where you do not own the car, you can order a limousine or a truck to solve your problem but surely do not need to own them or maintain their well-beings as you have to when you own the car. Similarly, private cloud owners can customize the compute, storage and networking components to best suit the specific system requirements within the limits of their cloud infrastructure, and public cloud is not customizable.

Finally, a last result from the car analogy is that you need to know how to drive a car if you are the owner and the driver of your car. Similarly, in private cloud, the owner of the cloud needs to know how to configure and manage the underlying cloud infrastructure. On the other hand, if you use a taxi (or a bus), you do not need to know how to drive or shortcuts to escape the city traffic but enjoy the ride. Likewise, a public cloud owner leaves the control of the infrastructure to the cloud provider and focuses on the core competency of the business.

To sum up, public cloud model is found as the best deployment approach considering the requirements of the thesis application. The advantages and disadvantages of the public and private cloud models are discussed previously in this section to give a meaningful comparison. However, the four main criteria that make public cloud the best fit for the thesis are scalability, flexibility, low cost, and pay-per-use billing model.

3.4.3 Amazon Web Services

This section gives an overview of Amazon Web Services (AWS) and mentions briefly the services that are used in the thesis application.

Amazon Web Services (AWS) is a cloud computing platform offering solutions for computing, storing, and networking, at different layers of abstraction. Similar cloud platforms include Google Cloud[51] and Microsoft Azure[52]. The term “web service” in Amazon Web Services means the provided cloud services can be controlled via a web interface [39].

The author has been using AWS services for the internal projects and supporting cloud customers at the company[53] that she works for and has a good experience with the AWS services overall. Although the author’s prior knowledge of AWS was a factor while choosing the cloud platform, it was not the sole reason. The thesis focuses on automating database cluster upgrades in the cloud environment. As declared in the first chapter, Ansible is chosen as the automation tool for implementing the solution. Ansible has many built-in modules (simple programs that are specialized for solving specific problems) and these modules are the building blocks to use when developing automated solutions using Ansible. For this reason, Ansible cloud modules carry a great value to develop an automated solution in the cloud. A comparison between cloud modules[54] shows that Ansible has 100 AWS modules while both Google Cloud and Microsoft Azure have less than 20 modules. As a result, AWS is found more practical while using Ansible to automate the cloud processes.

Amazon Web Services officially launched[55] in 2006 by announcing their first service Amazon Simple Storage Service (Amazon S3). Initially, Amazon S3 was a service that offered developers a storage infrastructure via an application programming interface (API). In 2015, The Company’s Letter to Shareholders [40], it is claimed that AWS is used by more than 1 million customers from organizations of every size across nearly every industry. Same report points that AWS announced 722 new features and services in 2015, a %40 increase over 2014. AWS shows prominent results as a fast-growing platform and also an innovative nature with new services and features overall the years. Being the pioneer of the cloud computing technology, AWS is considered the most mature cloud provider by the study results of the famous technological research company Gartner[56] [41]. According to the same study, AWS has the richest array of IaaS and PaaS capabilities, and it provides the deepest capabilities for governing a large number of users and resources comparing to its all competitors.

In the thesis, Amazon Elastic Compute Cloud (Amazon EC2)[57] service is used to launch virtual servers in the cloud and Amazon Virtual Private Cloud (Amazon VPC)[58] service

[51] https://cloud.google.com/
[52] https://azure.microsoft.com/en-us/
[53] https://www.2ndquadrant.com/en/
[54] http://docs.ansible.com/ansible/list_of_cloud_modules.html
[55] http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=830816
[56] http://www.gartner.com/technology/home.jsp
[57] https://aws.amazon.com/ec2/
[58] https://aws.amazon.com/vpc/

is used to create a virtual network for the EC2 instances. Amazon VPC service allows selecting an IP address range, creating subnets, configuring route tables and security groups, defining network gateways to the internet or a VPN endpoint, hence the service gives a complete control over the virtual networking environment that is built for the thesis application. As a side note, services on AWS are charged for on a pay-per-use pricing model.

AWS offers different data centers that are distributed in North America, South America, Europe and Asia Pacific. As of April 2017, the AWS Cloud operates 42 availability zones within 16 geographic regions around the world[59]. In AWS terminology, a region is a physical location in the world where AWS has multiple availability zones. Availability zones consist of one or more discrete data centers to provide better high availability, fault tolerance, and scalability than a single data center.

Globally distributed architecture of AWS Cloud has many benefits for AWS customers. First, this allows AWS customers to build business and serve their clients worldwide easily by only choosing a region (i.e. EU (Frankfurt), Asia Pacific (Tokyo), and South America (São Paulo)) for their AWS service. Most Amazon Web Services (i.e. Amazon EC2, Amazon DynamoDB, Amazon SQS) offer a regional endpoint (a URL that is the entry point for a web service) to reduce data latency in the applications distributed globally. Second, it is possible to avoid service outages (AWS provides a service level agreement[60] of 99.95 percent per zone that equates roughly 4 hours a year) with cross-region and cross-zone deployments of the applications. When the architecture is anticipated and designed for failure by multiple regions and zones, the possible outages will not impact the availability of the applications.

3.5 Configuration Management and Automation with Ansible

Ansible[61] is a configuration, and IT automation tool. This section gives an overview of Ansible and discusses why Ansible is chosen to work with over other alternatives such as Puppet[62], Chef[63] or SaltStack[64].

Among many other configuration management tools available, Ansible has some advantages. Primarily, Ansible has been designed to make configuration easy in many ways, from its choice for YAML[65] as its configuration management language to its ease

[59] https://aws.amazon.com/about-aws/global-infrastructure/
[60] https://aws.amazon.com/ec2/sla/
[61] https://www.ansible.com/
[62] https://puppet.com/
[63] https://www.chef.io/chef/
[64] https://saltstack.com/
[65] YAML is a language that approaches plain English http://yaml.org/

of setup without requiring any agents on the machines but only a working SSH connection.

Even though Ansible is one of the newer players in the configuration management tool market, it is very popular compared to its main competitors. Table 1 compares the popularity[66] of the tools based on their project profiles on source-code sharing platform Github. According to the data, Ansible is highly favoured by Github users with the maximum numbers of stars (rating mechanism by users), the highest numbers of project forks (copy of a repository) and the most watched (subscribed to the changes via notifications) project among other alternatives. The Ansible project has also an active community with the largest number of contributors and the second best ratio of releases per year after Chef Project.

Configuration management and IT automation tools can be classified in many ways, but it is possible to divide them into two architecture groups based on how they propagate the configurations: agent-based and agentless architectures. As it is shown in Table 1, Ansible supports agentless architecture, unlike other alternative products are chosen to compare such as Puppet, Chef, and SaltStack.

Agent-based systems have two different components: a server and a client called agent. The central server (usually called as master) contains all the configuration for the whole infrastructure, while the agents will contact to the master server to check if there is a new configuration for its machine is present. If there is a new configuration is present, the client will download it and apply it. As a consequence, agent-based architectures mostly implement pull-based configuration propagation model. In contrast, Ansible is push-based by default. As soon as the user (i.e. system administrator, software developer) runs the playbook (configuration scripts), Ansible connects to the remote server(s), executes the modules and changes the server state to a new state. This approach gives more control over when the changes will be applied to the servers, unlike waiting for the changes will be detected by the clients as in pull-based mechanisms. As can be seen in Table 1, Ansible also has official support for pull mode.

In agentless Ansible, there is no notion of a master node and children nodes (i.e Puppet master and agents, Chef server and clients, SaltStack master and minions), and no specific agent is present. SSH (Secure Shell) protocol is used for communications between the servers. Integrating an agentless system in an existing infrastructure is easier since it will be seen by the clients as a normal SSH connection and therefore no additional configuration is needed [42].

An agent-based system, like any other centralized system, involves certain availability and security concerns. First of all, the central machine that keeps all the configurations could easily be the single point of failure for the whole system. Secondly, since all

[66] Github popularity and activity data populated on May 6, 2017

machines have to be able to initiate a connection to the master machine, this machine could be attacked more easily than in an agentless case where the other machines do not have to be connected to the control machine. In agentless Ansible, control machine can remotely push the changes to all machines over SSH connection. In this scenario, control host could be placed behind a firewall that will block any incoming connection and reduce the attack risk.

Each config management tool appeals different IT experts from different backgrounds with different skillsets. Puppet and Chef require the knowledge of Ruby software language because both of them using a Ruby domain-specific language (DSL). To be productive with Puppet and Chef, one should know how to code in Ruby that eliminates System Administrators who do not necessarily have to have development skills to pursue their operations. Ansible is written in Python software language, but there is no need to know Python to use Ansible unless one needs to write her own module.

Anyone who is familiar with basic Linux (or any other system) system administration tasks can be productive with Ansible without any prior development skills. For example, if a person knows how to connect to a remote machine using SSH, install software packages, scripting and using basic commands via bash command-line shell, start and stop services, check and set file permissions and set environment variables, or familiar with these concepts then ready to use Ansible. Ansible uses YAML (data format language that was designed to be easy readable by humans) for configuration management and Jinja2 for templates. YAML is simple and intuitive for anyone who can write in English and Jinja2 templating language is well documented to check anytime it is needed.

Before automation tools are available, configuration management meant mainly editing configuration files manually. System administrators were writing lots of scripts, connecting to multiple servers to apply these scripts on each of the servers repetitively. Since these operations were mostly manual and manual tasks are known for their error-prone nature, they lead to heterogeneous and difficult to manage environments. Recent developments in cloud computing and the high rates of data growth created the need to manage infrastructure at a scale. For this reason, automation of the dynamic, complex infrastructures has become a necessity at every stage of the operations. The role of Ansible and other related configuration management and IT automation tools (i.e. Chef, Puppet, SaltStack) is helping to build the infrastructure as code as any mission-critical software in the system.

The other important detail of how different deployment models behave is that the push model is more suited for orchestration of a series of actions in specific order between multiple servers. For that type of orchestration, it is important to have control process which is aware of state on every machine and is able to make decisions about what to do next and where. As Ansible uses centrally controlled push model it fits the requirements to orchestrate the whole upgrade process well.

Table 1. Configuration Management Tools Comparison.

	Ansible	**Puppet**	**Chef**	**SaltStack**
Architecture	Agentless	Agent-based	Agent-based	Agent-based
Initial Release	2012	2005	2009	2011
Written in	Python	Ruby	Ruby & Erlang	Python
Appealing to	System Administrators	Software Developers	Software Developers	System Administrators
Learning Curve	Easy to Start and Develop Further	Steep Learning Curve	Steep Learning Curve	Easy to Start
Terminology	Directive: Task Script: Playbook Master: Control Machine Children: Hosts	Directive: Resource Script: Manifest Master: Master Children: Agents	Directive: Resource Script: Recipe (plural Cookbook) Master: Server Children: Client	Directive: State Script: SLS Formula (SLS SaLt State) Master: Master Children: Minions
Configuration Language	YAML	Puppet DSL, Ruby DSL	Ruby DSL	YAML
Template Language	Jinja2	EPP (Embedded Puppet)	ERB (Embedded Ruby)	Jinja2
Communication	SSH	Agents	Agents	Agents or SSH
Model	Push-based supports pull mode	Pull-based	Pull-based	Push-based
Remote Execution	Easy, built-in	Challenging, Puppet Enterprise provides built-in tools	Challenging (Knife Tool)	Easy, built-in
Execution Order	Sequential	Random by default requires explicit	Sequential	Sequential by default supports requisites
Ad-hoc Task Execution	Simple	Not supported	Not supported	Simple, allows execution over SSH
Enterprise Offering	Ansible Tower	Puppet Enterprise	Chef Automate, AWS OpsWorks, Hosted Chef	SaltStack Enterprise
Graphical User Interface	Offered with Ansible Tower	Offered with Puppet Enterprise	Offered with Chef Automate	Offered with SaltStack Enterprise
Popularity on Github	Stars: 22961 Forks: 7615 Watchers: 1635	Stars: 4439 Forks: 1836 Watchers: 496	Stars: 4810 Forks: 2002 Watchers: 410	Stars: 7654 Forks: 3561 Watchers: 563
Activity on Github	Contributors: 2665 Commits: 30031 Branches: 51 Releases: 151	Contributors: 442 Commits: 24857 Branches: 11 Releases: 303	Contributors: 509 Commits: 19236 Branches: 207 Releases: 954	Contributors: 1793 Commits: 81656 Branches: 17 Releases: 146

In conclusion, there are many reasons explained in this section why Ansible is chosen over the alternatives to automate database upgrades and the cloud infrastructure underneath. The main reasons include the ease of learning and installing Ansible, the large number of built-in modules, popularity and active community of Ansible, agentless architecture and push-based model.

4 Automated Upgrades in PostgreSQL

Availability of the system that is provided to customers is one of the indicators of a success and profitability of a company. Providing high availability can be (and usually is) achieved by adding redundancy to all components in the system (application servers, databases, etc). Here, automation is important for two reasons. First, having redundancy itself is only part of the equation, there is also need for switching the users (be it end users or applications) to the active server when the one they have been connected to fails. This can, of course, be done manually but automated systems will often detect problems and do the switch much faster and more reliably. The second reason why automation is important is the fact that having more and more redundancy means that the system also has more and more components that need to be managed and their configuration needs to be kept in a synchronized state. The repeatability is very important in this case (the redundant servers should be as close to each other as possible so that one can replace the other), so it is the ideal place where to automate because repeatability of results is one of the areas where software automation excels.

People who have the expertise on the system and operations area include system and database administrators, developers, or support engineers should put their knowledge into automating the processes and provide the team with a greatly simplified method of executing upgrades, ideally as simple as running a command or two. Moreover, automated tasks are easy to track, mainly because they apply “infrastructure as code” approach and the tasks can put through the source code versioning systems. The outcome of the automated tasks is clear and any team member can run the scripts without the requirement of becoming deeply familiar with all the little nuances of the upgrade procedure. Less time spent on upgrades leaves more time for the tasks that add the most value to the software projects.

4.1 Architecture and Implementation of the Automated Upgrade Platform

This section of the thesis gives an overview of how the Automated Cluster Upgrade Platform works. The platform is primarily designed to automate upgrades of PostgreSQL clusters in cloud. In addition to database upgrades, the platform allows provisioning cloud instances in a simple, customizable and more importantly automated way.

In the rest of this thesis, the Automated Cluster Upgrade Platform will be referred to as “pglupgrade” [43] tool. The “pglupgrade” name is derived from “(P)ost(g)reSQL (L)ogical (Upgrade)” phrase which emphasizes the logical replication feature of PostgreSQL that enables implementing an automated platform to achieve minimal downtime upgrades. Pglupgrade automates PostgreSQL cluster upgrades by utilizing pglogical (logical replication extension) and pgbouncer (connection pooler for PostgreSQL) tools with Ansible.

Pglupgrade tool is developed using Ansible. To be able to explain the platform design in details, key development concepts of Ansible including tasks, modules, playbooks, and plays should be described first. In Ansible, playbooks are the main scripts that are developed to automate the processes such as provisioning cloud instances and upgrading database clusters.

Playbooks may contain one or more plays. For example, the main playbook (shown in Appendix 1) of the pglupgrade tool, that is written to organize upgrade process, has eight plays to perform specific tasks on different host (server) groups. Playbooks may also contain variables, roles, and handlers in defined. Further explanations of these concepts can be found in the presentations [44] [45] [46] that the author herself delivered at the conferences include PostgreSQL Conference Europe 2015[67], FOSDEM PGDay 2016[68], and 5432...MeetUs! 2016[69]. In addition, she published articles [47] [48] about these concepts in her company blog as well.

The Pglupgrade tool consists of two main playbooks. The first playbook is provision.yml that automates the process for creating Linux machines in cloud, according to the specifications. The second playbook is pglupgrade.yml that automates upgrade process of database clusters. This chapter explains pglupgrade playbook (pglupgrade.yml), hence it is the primary focus of the thesis. Provisioning playbook (provision.yml) is explained in Section 4.2.1.

Pglupgrade playbook has eight plays to orchestrate the upgrade. Each of the plays uses one configuration file (config.yml), perform some tasks on the hosts or host groups that are defined in host inventory file (host.ini).

An inventory file lets Ansible know which servers it needs to connect using SSH, what connection information it requires, and optionally which variables are associated with those servers Figure 6 shows a version (with a set of values) of the pglupgrade host.ini file that has been used to execute automated cluster upgrades for one of the case studies.

The sample inventory file that is illustrated in Figure 6 contains five hosts under five host groups that include old-primary, new-primary, old-standbys, new-standbys and pgbouncer. A server could belong to more than one group. For example, the old-standbys is a group containing the new-standbys group, which means the hosts that are defined under the old-standbys group (54.77.249.81 and 54.154.49.180) also belongs to the new-standbys group. In other words, the new-standbys group is inherited from (children of) old-standbys group. This is achieved by using the special ":children" suffix.

[67] October 27-30, 2015 Vienna Austria https://2015.pgconf.eu/
[68] January 29, 2016, Brussels Belgium https://fosdem2016.pgconf.eu/
[69] June 28-29, 2016, Milan Italy http://2016.5432meet.us/en/home-en/

```
[old-primary]
54.171.211.188

[new-primary]
54.246.183.100

[old-standbys]
54.77.249.81
54.154.49.180

[new-standbys:children]
old-standbys

[pgbouncer]
54.154.49.180
```

Figure 6. Configurable Host Inventory File (host.ini).

Once the inventory file is ready, Ansible playbook can be run via ansible-playbook command, shown in Figure 7, by pointing to the inventory file (if the inventory file is not located in default location otherwise it will use the default inventory file).

```
$ ansible-playbook -i hosts.ini pglupgrade.yml
```

Figure 7. Running an Ansible Playbook.

As mentioned earlier, pglupgrade.yml file is the main playbook in the pglupgrade tool that organizes the actual upgrade operation. This playbook uses a configuration file (config.yml) that allows users to specify values for the logical upgrade variables.

The configuration file (config.yml) that is shown in
Figure *8*, stores mainly PostgreSQL-specific variables that are required to setup a PostgreSQL cluster such as “postgres_old_datadir” and “postgres_new_datadir” to store the path of the PostgreSQL data directory for the old and new PostgreSQL versions; “postgres_new_confdir” to store the path of the PostgreSQL config directory for the new PostgreSQL version; “postgres_old_dsn” and “postgres_new_dsn” to store the connection string for the “pglupgrade_user” to be able connect to the “pglupgrade_database” of the new and the old primary servers. Connection string itself is comprised of the configurable variables so that the user (“pglupgrade_user”) and the database (“pglupgrade_database”) information can be changed for the different use cases.

As a key step for any upgrade, the PostgreSQL version information can be specified for the current version (“postgres_old_version”) and the version that will be upgraded to (“postgres_new_version”). In contrast to physical replication where the replication is a copy of the system at the byte/block level, logical replication allows selective replication where the replication can copy the logical data include specified databases and the tables in those databases. For this reason, config.yml allows configuring which database to replicate via “pglupgrade_database” variable. Also, logical replication user needs to have replication privileges, which is why "pglupgrade_user" variable should be specified in the configuration file. There are other variables that are related to working internals of pglogical such as "subscription_name" and "replication_set" which are explained in details in Section 4.2.3 while describing pglogical role.

```
ansible_user: admin

pglupgrade_user: pglupgrade
pglupgrade_pass: pglupgrade123
pglupgrade_database: postgres

replica_user: postgres
replica_pass: ""

pgbouncer_user: pgbouncer

postgres_old_version: 9.5
postgres_new_version: 9.6

subscription_name: upgrade
replication_set: upgrade

initial_standbys: 1

postgres_old_dsn: "dbname={{pglupgrade_database}} host={{groups['old-
primary'][0]}} user={{pglupgrade_user}}"
postgres_new_dsn: "dbname={{pglupgrade_database}} host={{groups['new-
primary'][0]}} user={{pglupgrade_user}}"

postgres_old_datadir: "/var/lib/postgresql/{{postgres_old_version}}/main"
postgres_new_datadir: "/var/lib/postgresql/{{postgres_new_version}}/main"

postgres_new_confdir: "/etc/postgresql/{{postgres_new_version}}/main"
```

Figure 8. Configuration File (config.yml).

Pglupgrade tool is designed to give the flexibility in terms of High Availability (HA) properties to the user for the different system requirements. The "initial_standbys" variable is the key for designating HA properties of the cluster while the upgrade operation is happening. For example, if "initial_standbys" is set to 1 (can be set to any number that cluster capacity allows), that means there will be 1 standby created in the upgraded cluster along with the master before the replication starts. In other words, if you have 4 servers and you set initial_standbys to 1, you will have 1 primary and 1 standby server in the upgraded new version, as well as 1 primary and 1 standby server in the old version. This option allows the reuse the existing servers while the upgrade is still happening. In the example of 4 servers (as demonstrated in first use case), the old primary and standby servers can be rebuilt as 2 new standby servers after the replication finishes.

When “initial_standbys” variable is set to 0, there will be no initial standby servers created in the new cluster before the replication starts.

Finally, configuration file allows specifying old and new server groups. This could be provided in two ways. First, if there is an existing cluster, IP addresses of the servers should be entered into hosts.ini file by considering desired HA properties while upgrade operation. The second way is to run provision.yml playbook to provision empty Linux servers in cloud (AWS EC2 instances) and get the IP addresses of the servers into the hosts.ini file. Either way, config.yml will get host information through hosts.ini file.

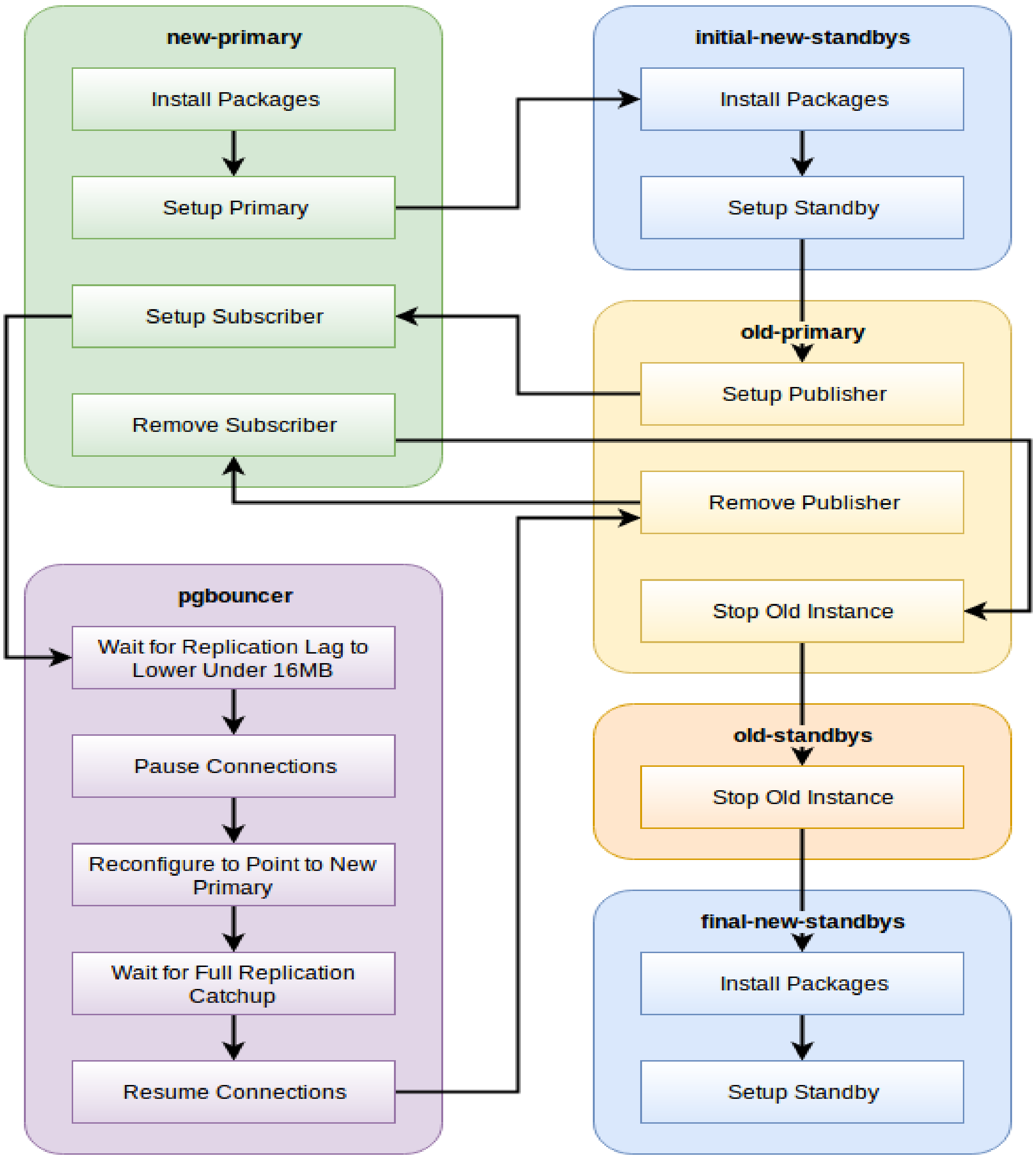

Figure 9. Workflow of the upgrade process.

After explaining the configuration file (config.yml) which is used by pglupgrade playbook. The workflow if the upgrade process is shown on Figure 9. There are six server groups that are generated in the beginning based on the configuration (both hosts.ini and the config.yml). The new-primary and old-primary groups will have always one server, pgbouncer group can have one or more servers and all the standby groups can have zero or more servers in them. Implementation-wise the whole process is split into eight steps. Each step corresponds to a play in the pglupgrade playbook, which performs the required tasks on the assigned host groups. The upgrade process is explained through following plays:

1. Build hosts based on configuration: Preparation play which builds internal groups of servers based on the configuration. The result of this play (in combination with the hosts.ini contents) are the six server groups shown on Figure 9 which will be used by the following seven plays.
2. Setup new cluster with initial standby(s): Setups an empty PostgreSQL cluster with the new primary and initial standby(s) (if there are any defined). Ensures that there is no remaining from PostgreSQL installations from the previous usage.
3. Modify the old primary to support logical replication: Installs pglogical extension. Then sets the publisher by adding all the tables and sequences to the replication.
4. Replicate to the new primary: Sets up the subscriber on the new master which acts as a trigger to start logical replication. This play finishes replicating the existing data and starts catching up what has changed since it started the replication.
5. Switch the pgbouncer (and applications) to new primary: When the replication lag converges to zero, pauses the pgbouncer to switch the application gradually. Then it points pgbouncer config to the new primary and waits until the replication difference gets to zero. Finally, pgbouncer is resumed and all the waiting transactions are propagated to the new primary and start processing there. Initial standbys are already in use and reply read requests.
6. Clean up the replication setup between old primary and new primary: Terminates the connection between the old and the new primary servers. Since all the applications are moved to the new primary server and the upgrade is done, logical replication is no longer needed. Replication between primary and standby servers are continued with physical replication.
7. Stop the old cluster: Postgres service is stopped in old hosts to ensure no application can connect to it anymore.
8. Reconfigure rest of the standbys for the new primary: Rebuilds the other standbys if there are any remaining hosts rather than initial standbys. In the second case study, there are no remaining standby servers to rebuild. This step gives the chance to rebuild the old primary server as a new standby if pointed in the new-standbys group at hosts.ini. The reusability of existing servers (even the old primary) is achieved by using the two-step standby configuration design of the pglupgrade tool. User can specify which servers should become standbys of the new cluster before the upgrade and which should become standbys after the upgrade.

4.2 Ansible Roles

Playbooks organize tasks and roles organize playbooks. Even though it is possible to write all the automation tasks in one big playbook, it may result in an unmaintainable solution in long term. Modern IT systems have many moving parts that require modularity in automated scripts. Instead of repeating the same tasks (or config files, templates, dependencies, variable files) in many playbooks; wrapping them as roles and calling these roles from wherever necessary, will make playbooks efficient, easy to understand, debug and maintain. This way, contents can be easily shared between playbooks, without rewriting the same components over and over.

Pglupgrade tool uses roles to organize playbooks efficiently. The tool contains four main roles include AWS, Postgres, Pglogical, and PgBouncer. This section discusses these roles and their sub-roles by explaining key tasks and modules.

4.2.1 AWS Role

Pglupgrade tool is developed to automate database upgrades in cloud environments. Therefore, the platform is tested in cloud, specifically on AWS EC2 instances. The tool provides optional provisioning solution via provisioning playbook (provision.yml, see Appendix 5). The provisioning playbook creates empty Linux machines that are configured according to the specifications such as which Linux distribution will be used, which SSH keys will be installed, in which cloud region the machine(s) will be installed, how many machines will be provisioned, what are the disk sizes, memory and storage limits, what is the name of the VPC, subnet info etc. If the user already has the machines, there is no need to run the provisioning playbook, but this is the easiest way to set up the machines that fit the system requirements that are defined in config-aws.yml.

Provisioning playbook contains aws/provision role (shown in Appendix 6) to accomplish provisioning of the cloud instances. The aws/provision role consists of tasks that are divided into three files: main.yml, ami.yml and vpc.yml. The main.yml is the primary task file that includes ami.yml for the tasks related to Amazon Machine Images (AMIs), and vpc.yml for the tasks related to Amazon Virtual Private Cloud (Amazon VPC). The role performs the following tasks on the designated hosts:

- Ensure the SSH key is present: Makes sure that the user’s SSH key, which will be used to access the servers, is present in every AWS region defined in the config-aws.yml.
- Configure VPCs (in vpc.yml): Ensures VPC is present (creates the VPC if it does not exist), configures the subnet of the VPC, creates internet gateway and route table for the VPC. Finally, it creates security groups and enables access to SSH and Postgres ports.
- Configure AMIs (in ami.yml): Finds the AMI (is a virtual image that is used to create a virtual machine within the Amazon EC2) that is given in aws-config.yml

and registers this AMI with the other useful AMI-related information as a fact for the playbook.

- Ensure EC2 instances and volumes are present: Checks if all the servers present, creates empty Linux servers if they do not exist and configures them based on the given specification in config-aws.yml.

4.2.2 PostgreSQL Role

Pglupgrade tool contains a PostgreSQL role (see Appendix 2) that has five sub-roles for automating PostgresSQL-specific tasks. PostgreSQL sub-roles include postgres/conf, postgres/pkg, postgres/primary, postgres/remove and postgres/standby roles that are explained in the following list:

1. postgres/pkg: Installs PostgreSQL packages automatically based on the new PostgreSQL version specified in the config.yml.
2. postgres/conf: Ensures the PostgreSQL configuration directory exists (and creates if it does not exist) for the new PostgreSQL cluster. Sets up two main configuration files namely postgresql.conf and pg_hba.conf via matching jinja2 templates. Users can modify these two configuration files but default values are ready to support logical replication.
3. postgres/primary: Creates the PostgreSQL instance that will be the new primary server. Ensures that the database that is being upgraded and the user that is used by standby servers exists. Additionally, this role utilizes several Ansible PostgreSQL modules (i.e. postgresql_db: creates database, postgresql_user: creates user).
4. postgres/standby: Creates a base backup of the new primary server and starts the new standby servers based on this backup.
5. postgres/remove: Stops postgres service and cleans all PostgreSQL installations.

4.2.3 Pglogical Role

Pglupgrade tool utilizes Pglogical [49] extension as its upgrade method and contains a Pglogical role (see Appendix 3) for automating Logical Replication processes. The Pglogical role has four sub-roles include pglogical/common, pglogical/publisher, pglogical/subscriber and pglogical/cleanup roles that are explained in the list below:

1. pglogical/common: Establishes the common logic between publisher and subscriber nodes. Installs the pglogical extension, creates the temporary logical replication user with the correct permissions (i.e. SUPERUSER, REPLICATION, LOGIN). Additionally, this role utilizes postgresql_ext module to add the pglogical extension to the pglupgrade database.
2. pglogical/publisher: Creates the pglogical publisher node. Then, creates the replication set which is used in the upgrade. Lastly, adds all the tables and the sequences to the replication set. An important detail is here that the tables are added one by one to the replication set. This sequential approach is applied to

avoid possible deadlocks for running transactions. Because the order of table locking might be different in the software application and this method eliminates the deadlock risk by considering the possibility.

3. pglogical/subscriber: Creates the pglogical subscriber node. Then, creates the subscription. Finally, waits for the subscription to be ready. This step takes some time (depending on the data size) because it copies all the existing data.
4. pglogical/cleanup: First, ensures the subscription does not exist by dropping it if exists. Next, ensures the local node does not exist by dropping it if it is present. Then, removes the pglogical extension and drops the temporary pglogical user.

4.2.4 PgBouncer Role

Pglupgrade tool has a PgBouncer role (shown in Appendix 4) for automating the tasks related with PgBouncer.

The PgBouncer role starts with waiting for the replication lag to drop under 16 MBs. This buffer is needed because it might not be possible to reach zero difference while there are still data being written by the active database sessions. After the minimal lag is achieved, the next task pauses PgBouncer. Already running transactions will finish but the new ones will be queued. While PgBouncer is paused, the configuration file of PgBouncer (pgbouncer.ini) is updated to point to the new primary server and is reloaded. Then, the next task waits until the replication difference gets to the zero because now it is possible to catch up the replication since there are no writes happening (PgBouncer was paused and the transactions were queued). Once this happens, PgBouncer is resumed. Finally, all the waiting transactions are directed to the new primary server and start running on there.

4.3 Implementation Limitations

It is worth noting that the upgrade mechanism as implemented will not work on every PostgreSQL cluster. One example of where the pglupgrade would not work well is visible in the figures showing the impact of initial data copying on the old primary server. In case there are no spare resources on the primary server, the pglupgrade would limit the usability of the server or possibly completely overload it when copying the initial data. It is the opinion of the author that logical replication implementations (such as Pglogical extension) should have the option for limiting the speed of the initial data copy so that the impact on the server can be reduced at the price of longer overall time needed for the upgrade.

It is also possible to have PostgreSQL cluster which has so many writes that the logical replication will never finish the catchup phase because of the growing backlog of data to be applied.

Another potential problem is that pglogical can only replicate tables which have primary keys or tables without primary keys which are not subject to updates or deletes (insert-only tables). This limitation is however removed in the upcoming PostgreSQL 10.

Finally, neither pglogical nor the PostgreSQL logical replication support transparent replication of DDL commands so applications cannot run those commands while the upgrade is running. In practice, DDL commands are only needed during deployment of a new version of the application (database) so this requirement is easy to enforce.

4.4 Applicability to Other Systems and Environments

The software chosen is all open source and will work same in traditional data centers on both bare metal and virtual machines as well as in other cloud systems. With the exception of pglupgrade Ansible script, all of the components will work on Windows operating system as well.

The proposed upgrade method will work on most databases that allow logical replication across major versions. These include MySQL with the binlog replication or Oracle using GoldenGate.

One component missing in most other database management system is pgbouncer which acts as connection proxy with knowledge of the protocol and ability to pause connections, making the switch of servers completely transparent to the application which sees it as a just short temporary slowdown. But even then minimum downtime can be achieved using virtual IP, DNS, or some other TCP connection proxy software. The main difference between using protocol aware proxy like pgbouncer and other methods to switch applications to new cluster is that with pgbouncer the application only sees increased latency for the queries while with the other methods it will get connection errors during the few seconds it takes to switch.

5 Case Studies for Automated PostgreSQL Upgrades

To evaluate the proposed approach, it was tested in practice with two different cluster configurations that serves two different purposes. The upgrade tools provided as part of PostgreSQL were used for comparison with the approach proposed in this thesis.

It is necessary to simulate the impact of the upgrade on the application(s) connected to the database cluster. The sample application chosen for this purpose was pgbench[70] which is the standard benchmarking tool included in PostgreSQL. Pgbench simulates (O)n(L)ine (T)ransaction (P)rocessing (OLTP) application and is loosely based on the TPC-B[71] benchmark specification. In default settings, pgbench will try to execute as many transactions as possible and generate reports about how many transactions per second were possible and also latencies of given transaction. This allows us to see the impact of the upgrade on the application's ability to serve requests. It also shows the impact of the online upgrade as proposed in this thesis on a fully loaded server.

5.1 Impact Analysis of PostgreSQL Upgrades

Upgrading to a new major version is a task which can require considerable preparation over total execution time. Upgrade planning takes a lot of time with ecosystem discussions, an order of planned processes, different scenarios for minimizing the downtime which usually extends into multiple long meetings. Automation of upgrade process is a clear benefit which will reduce the stress of a large regular process for many companies and time-consuming meetings will leave its place to efficient evaluation meetings.

Traditional methods of upgrading PostgreSQL major version require the cluster to be shut down during the process of the upgrade. This presents a problem for overall service availability. As a result, the upgrade usually requires long preparation and careful selection of the right time when to upgrade. Sometimes it is possible to use the standby servers to provide limited read-only service. This, however, complicates both the application development and the upgrade procedure as the application has to support read-only mode and the upgrade procedure has to include additional steps for the switch to read-only mode and back and reconfiguring the standby servers accordingly.

The upgrade procedure proposed in this thesis tries to avoid this problem completely. This is achieved by using the pgbouncer connection proxy which can move the application to the new server(s) transparently. The other important method for achieving service availability is the fact that the pglupgrade is capable of having standby servers running both for old and new server during the whole upgrade procedure. The result is that even

[70] https://www.postgresql.org/docs/devel/static/pgbench.html
[71] http://www.tpc.org/tpcb/default.asp

if some server fails, it is immediately possible to failover to the standby. This may in some situations result in failure to upgrade, but not in service interruption. The service availability is one of the main issues with the standard method of upgrading PostgreSQL which is impossible to completely avoid without a radical change in the procedure. The case studies demonstrated a way for improving the situation and making major version upgrades a less scary proposition for the organizations that use PostgreSQL.

5.2 Setting up the Environment and Choosing Software Version

For most of the software used in the testing, the chosen version was the latest stable release available at the time of running the tests (April 2017). The exception is PostgreSQL, where both previous major version and current major version are used as the test, is supposed to demonstrate the version upgrade. The only important limitation in terms of software version for testing is the need to use PostgreSQL 9.4 or higher as that is the version which added support for logical decoding feature which Pglogical uses to implement replication. The actual versions of software used are following:

- Ubuntu 16.04
- Ansible 2.3
- Pgbouncer 1.7.2
- PostgreSQL 9.5.6
- PostgreSQL 9.6.1
- Pglogical 2.0.0

All tests were running on Amazon EC2. Each instance was running on a 64-bit system, had 2 Virtual CPUs, 4GB RAM for memory, and 110GB EBS for storage.

5.3 Defining Metrics

To evaluate the proposed approach against the ones provided as standard solutions by PostgreSQL, it is necessary to describe metrics which will be used for comparison.

The first main metric is the downtime of the primary server needed for the upgrade to complete. This translates to the time application cannot use the database at all. The second main metric is the time it takes for the upgraded cluster to reach original high availability and scalability properties of the old cluster. This is important because it maps directly to how long it takes until the application can serve requests at full capacity.

There are also some side metrics that can be also collected. Namely, the time it takes to partial HA (when it is safe to start using the cluster), the time it takes for the whole upgrade process to finish and any additional disk space required by the upgrade procedure is also measured.

Additionally, performance measurements of the cluster is taken during the Pglupgrade to judge feasibility of cluster use during the upgrade. For similar reasons the performance of the cluster is also measured during the additional standby creation.

5.4 Cluster Size Considerations

The size of a database cluster will affect the length of the upgrade process as well as network bandwidth and disk space required.

The instance size (the size of data directory on disk) directly affects the time needed to upgrade the database. Different methods of upgrade will be affected differently by various parts of what database stores in the data directory. The old traditional method of logical dump and restore will be mostly affected by the actual data size as will the proposed method of logical replication. The newer traditional method of upgrade, pg_upgrade is mainly affected by the size of metadata (the information about what tables and other user objects exist in the database and also the transaction status and data visibility information which PostgreSQL also stores).

Another cluster property which has an effect on the upgrade is the instance count (number of servers in the cluster). This affects how fast the upgraded cluster reaches high availability and transactional load targets.

5.5 Evaluation Setup

Both of the use cases have a similar setup of the initial cluster. The main difference is the cluster topology that is described in details for each case in Chapters 6.6 and 6.7.

The initial instances were provisioned using the provision.yml. However, the PostgreSQL instances, including configuration and replication setup, was done manually.

Once cluster was running the initial data was loaded using the pgbench with scaling factor of 2000 which produces the database size of around 27GB (1GB of database size is scaling factor of approximately 75 in pgbench). Database cluster with freshly loaded data does not represent real-life cluster well enough however because of PostgreSQL also stores per transaction information in various LRU (Least Recently Used) caches on disk and size of these affects the length of the upgrade. For this reason, application usage was simulated by running the pgbench in normal benchmark mode for the period of one hour. This generated additional 3 GB of the database size, primarily as part of the pgbench_history table where pgbench inserts new data during the benchmark run.

Once everything was done, the data directory of the to-be-upgraded cluster had 30GB. Before testing started a snapshot of the data directory was made so that the different upgrade solutions would have the same state of the database before the upgrade was done using each of them.

5.6 Evaluation Procedure

Each use case database was upgraded using three different methods.

First one is the pg_dump/pg_restore procedure, which is the standard logical backup and restore tooling for PostgreSQL which is the oldest method available for upgrading the PostgreSQL clusters.

The second method is pg_upgrade which does the conversion of binary data files from old major version to the new one. This tool is available as part of standard “contrib” modules of PostgreSQL.

And finally, the third method is the pglupgrade tooling introduced in this thesis. This is the only method of the three that keeps the cluster running during the whole upgrade. As such it is also important to monitor the transaction rate and the latency of queries that are being executed on the primary server during the upgrade itself. These help assessing the feasibility of the online upgrade process as too low transaction rate or too high latency would mean that the primary is not usable in practice during the upgrade.

5.7 First Case: Database with 3 standby servers used for high availability

For the first test case, a cluster with 1 primary server and 3 standby servers are chosen. The 3 standby servers are used for high availability purposes. This is a common setup which allows having standby servers available, even after the failover procedure was executed due to the failure of the primary server. It also allows having cluster spread across two regions so that failover can happen when the whole primary region has problems.

The setup used 5 AWS servers in total, 4 for the database cluster and 1 acting as a connection proxy using PgBouncer as described by Figure 10a. The standby servers from original cluster were reused by the new cluster to limit the cost while keeping some high availability requirements during the upgrade.

The changes in the cluster configuration done by Pglupgrade during the upgrade are presented on Figure 10. The initial state shown on Figure 10a is common for all three methods of upgrade. The intermediary step where there are two clusters connected using logical replication is visible on Figure 10b. The Figure 10c shows again two clusters but this time after the replication is cut and the applications are connected to the upgraded cluster. And finally the Figure 10d shows the final state of the cluster which is again common for all three upgrade methods.

In a real-life scenario, there would be 2 pgbouncer servers for high availability of the connection proxy as well. But since that does not change the upgrade process in any meaningful way (the only difference is that the step 3 needs to be done for more than one server) it is decided to not include this for budget reasons.

Table 2 presents the results for individual metrics defined in the Section 6.2. These numbers were collected based on following facts. It took 6 minutes and 47 seconds for pg_dump to dump the database and then 17 minutes and 40 seconds to restore it using pg_restore to the new version of PostgreSQL. The new cluster in case of pg_dump already had one standby ready so the restore was also replicated immediately, hence no difference of time between primary downtime and partial HA. It took approximately 38 minutes to clone one standby and in the case of pg_dump, two standbys were cloned in parallel to decrease the time needed for full cluster capacity.

The pg_upgrade did a copy of the data directory, which was slightly smaller as there was less per transaction metadata so the additional space needed was slightly less than the size of existing cluster. It managed to do so in 16 minutes and 25 seconds. As we were reusing the same old servers, the standbys did not need to be cloned from the scratch but rsync[72] utility was used to synchronize the data directory of the old standbys with the new primary (while the primary was still off) which took 12 minutes and 31 seconds for all servers in parallel. That means the upgrade with pg_upgrade was much faster than with pg_dump, although depending on the exact needs, the downtime as seen by the application could be slightly longer. Also, the total time spent upgrading was relatively short.

Table 2. Comparison of the Upgrade Methods (First Case).

Metric	**pg_dump/pg_restore**	**pg_upgrade**	**pglupgrade**
Primary downtime [hh:mm:ss]	00:24:27	00:16:25	00:00:03
Partial cluster HA [hh:mm:ss]	00:24:27	00:28:56	00:00:03
Full cluster capacity [hh:mm:ss]	01:02:27	00:28:56	00:38:00
Length of upgrade [hh:mm:ss]	01:02:27	00:28:56	01:38:10
Extra disk space	800 MB	27 GB	10 GB

[72] https://linux.die.net/man/1/rsync

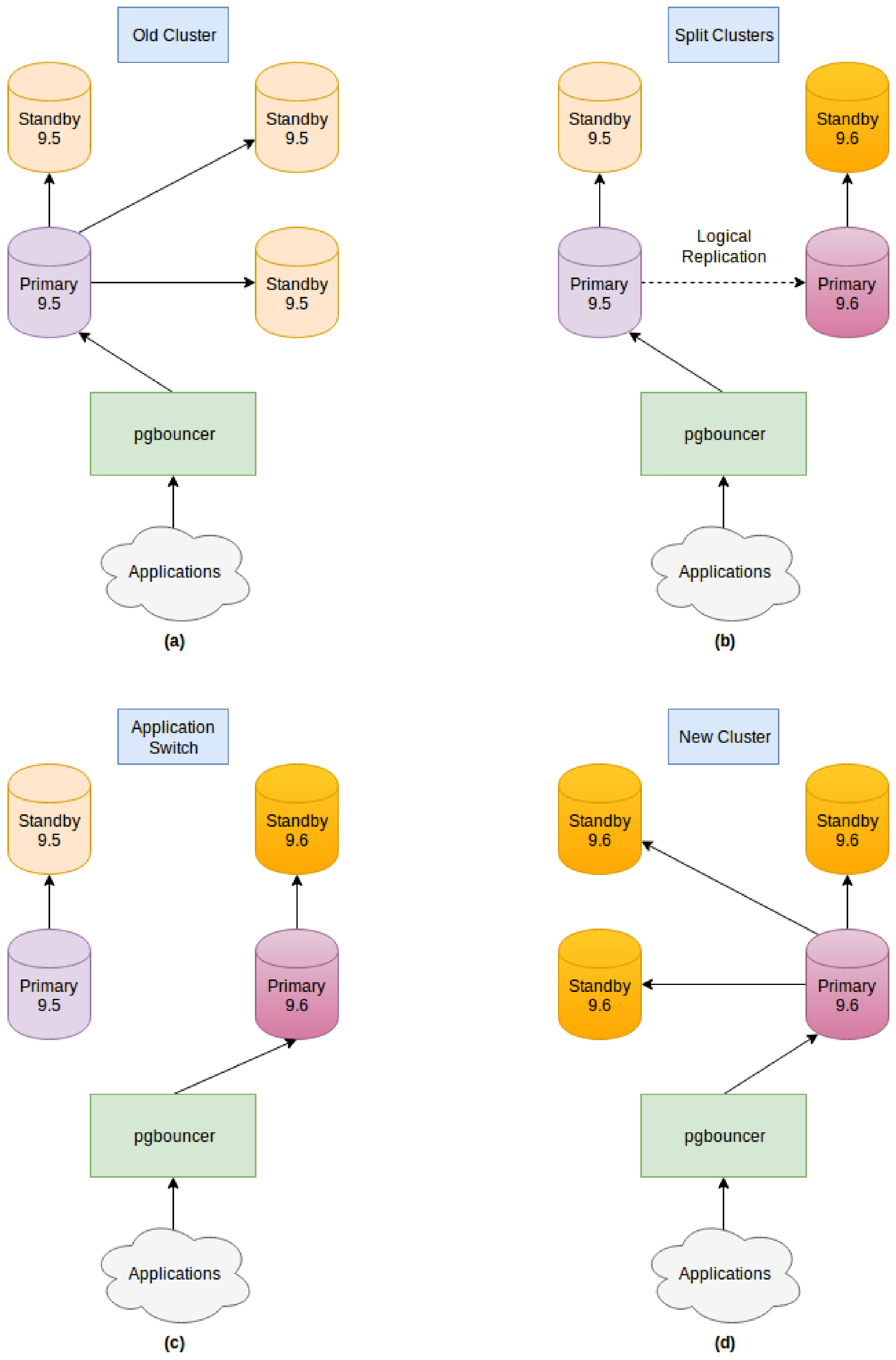

Figure 10. Four steps pglupgrade goes through during upgrade (first case).

Finally, the pglupgrade achieved minimal system interruption with just 3 seconds of downtime and this downtime was perceived by the application (pgbench) as delay in query response, no errors were returned to the application. Since pglupgrade created a new cluster with one standby similarly to pg_dump, the partial HA was again achieved immediately as the data in the new primary were replicated to the standby during the upgrade. The 10GB or extra space was needed to hold the write ahead log of PostgreSQL as new data were written but the logical replication needed to see historical records so that it could replicate everything correctly. The conclusion which can be drawn from this experiment is that the pglupgrade has indeed caused least disruption to the application (and users) of the three compared solutions, providing near-zero downtime upgrade.

5.8 Second Case: Database with 10 standby servers used for spreading the reads

The second case used a total of 23 servers. One was again reserved for pgbouncer to do the proxy. The old cluster comprised of one primary and ten standbys. This simulates the scenario where the application needs to do a high amount of reads and so many standby servers are used to satisfy the read scalability requirements. This time there was no reuse of the servers and the new cluster was created on freshly provisioned servers again with one primary and ten standbys. The reasoning for creating a fresh cluster is that given the requirements for read scalability, the cluster is not practically usable unless there are several standbys present and having freshly built cluster improves the speed of the upgrade.

This is also another example of advantages automated deployment in a cloud environment. Without the cloud, it would be necessary to either buy new servers or decommission the old ones once the upgrade is done, or prolong the upgrade by reusing the old servers like in the first use case. The automated setup makes it easy to create all these new instances without having to repeat the provisioning steps and configuring each server individually.

As with the previous use-case, there are four cluster configurations that the upgrade goes through as shown in Figure 11. The initial state before the upgrade (Figure 11a) and final state after the upgrade (Figure 11d) are again common for all three solutions. The intermediary steps are done by pglupgrade where there are two clusters connected using logical replication, and same two clusters after the replication are cut and the applications are connected to the upgraded cluster are visible on Figure 11b and Figure 11c respectively.

Table 3 presents the results for individual metrics defined in Section 6.2. These numbers were collected based on following facts. It took 6 minutes and 34 seconds for pg_dump to dump the database and then 17 minutes and 18 seconds to restore it using pg_restore to the new version of PostgreSQL. As the new cluster was on a completely new set of servers, it already had all standby servers ready so the restore was also replicated immediately, hence the time for downtime, HA, capacity and upgrade length are all the same.

The pg_upgrade as expected created a slightly smaller copy of the data directory, like was the case with the previous use case. It managed to do it in 17 minutes and 3 seconds. However, it needed to reclone all the new standbys which were done two at a time and took about 37 and half minutes per server. This resulted in very long time for the whole upgrade and for full cluster capacity and the application using this cluster for read scaling may need to wait for over 3 hours to work for all users.

Table 3. Comparison of the Upgrade Methods (Second Case).

Metric	**pg_dump/pg_restore**	**pg_upgrade**	**pglupgrade**
Primary downtime [hh:mm:ss]	00:23:52	00:17:03	00:00:05
Partial cluster HA [hh:mm:ss]	00:23:52	00:54:29	00:00:05
Full cluster capacity [hh:mm:ss]	00:23:52	03:19:16	00:00:05
Length of upgrade [hh:mm:ss]	00:23:52	03:19:16	01:02:10
Extra disk space	800 MB	27 GB	10 GB

Finally, the pglupgrade achieved minimal system interruption with just 5 seconds of downtime again without any errors show to the application, an only long time taken by some queries. Just like was the case with pg_dump, the pglupgrade created the new cluster with all 10 needed standby servers so the application using the cluster for scaling the reads would be able to run uninterrupted during the whole upgrade as well as after the upgrade was finished.

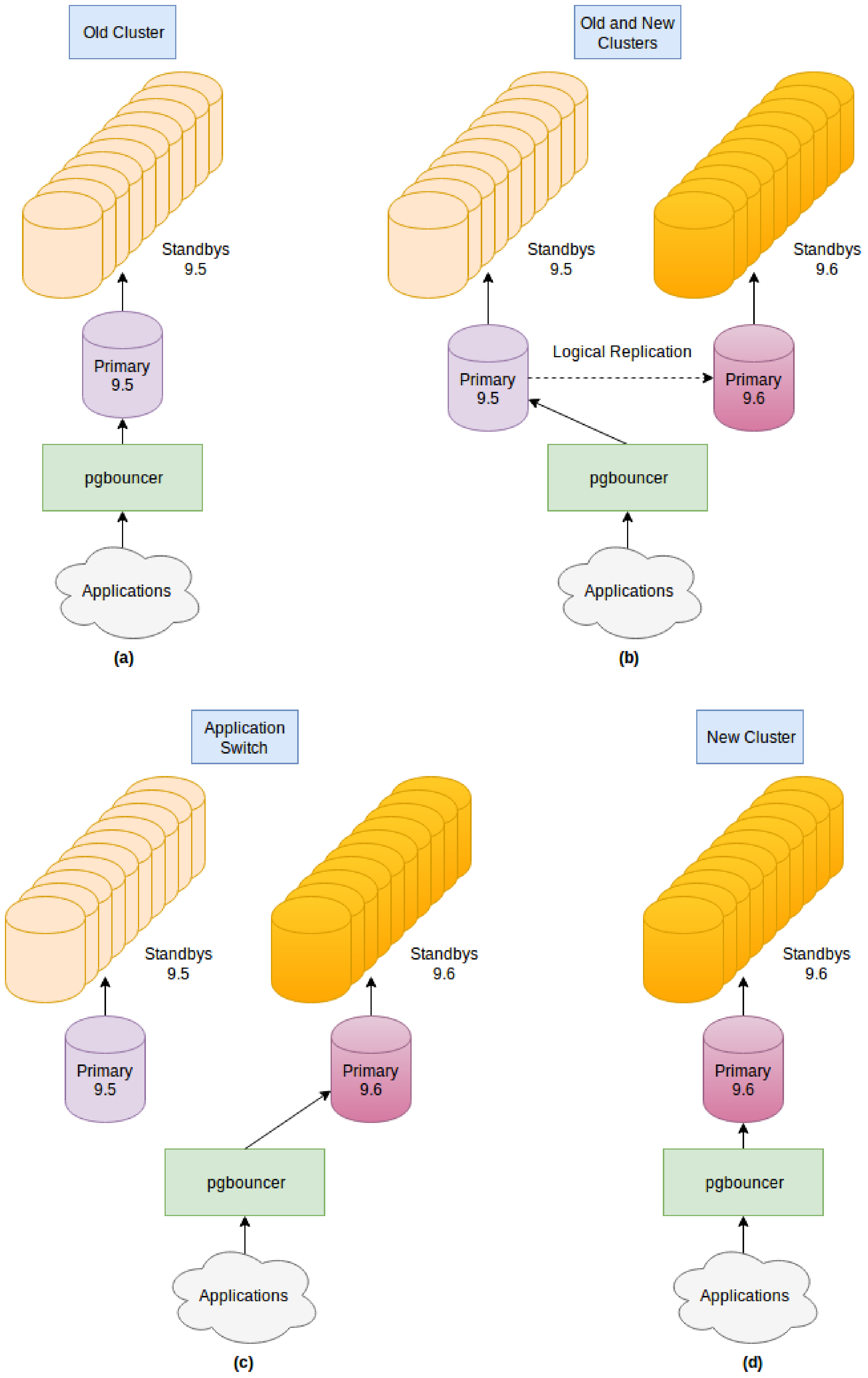

Figure 11. Four steps pglupgrade goes through during upgrade (second case).

5.9 Interpreting the Results

To interpret the results collected during evaluation of the two use-cases it is important to understand that the upgrade is happening in multiple phases each with different impact on the cluster. These steps are somewhat different based on which upgrade method is used.

For pg_dump and pg_restore method, the first phase is stopping the connections to the server, this can be done either on connection proxies like pgbouncer, or firewall or by shutting down the applications. The second phase is the actual dump of the data into a file. Afterwards, the data are restored to the new version of PostgreSQL.

In the case of pg_upgrade, the first phase is stopping the old server, followed by the run of pg_upgrade which copies the binary data to the new cluster and updates the system catalogs (metadata) accordingly.

These steps are interactive and there is not much one can monitor except the growing size of the data directory.

Once the initial upgrade of the primary server is done, either with pg_dump or pg_upgrade, the standbys are added. Cloning of standbys usually happens when the new primary is already being used to limit the downtime of the cluster to a minimum. It is important to monitor the state of the cluster while cloning is in progress because it affects the performance of the database. For this reason, it is not feasible to clone too many standby servers in parallel, but either serialize the process or limit the parallelization to the only couple of servers at a time so that the impact on performance is minimized. The impact of the performance is shown in Figure 12. It is visible that there is about 25% performance impact on the primary server in this test.

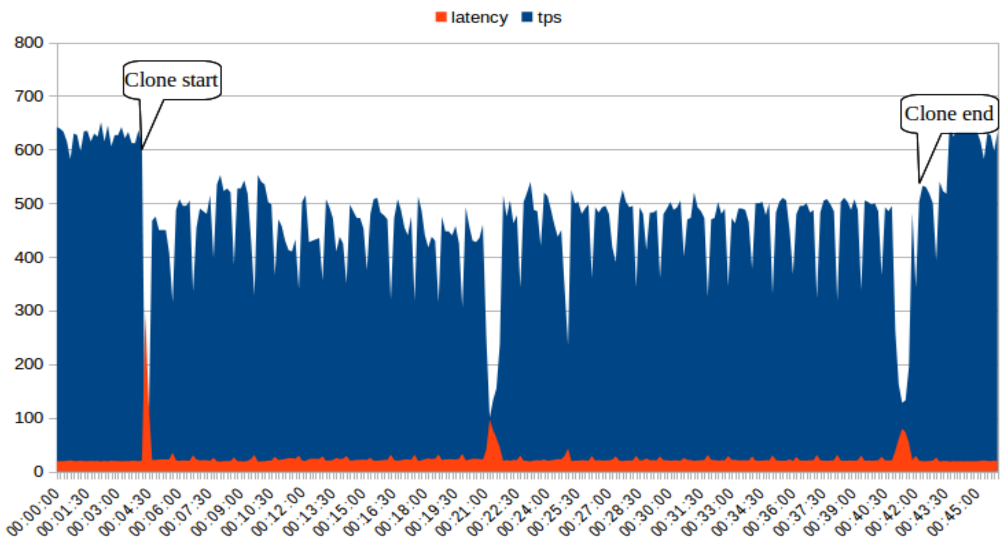

Figure 12. Transaction rate and latency graph during standby cloning process.

A similar situation occurs for pglupgrade already during the initial upgrade process. Since the cluster is being actively used while the upgrade is happening, the effect of the upgrade needs to be monitored as well. The pglupgrade has multiple, more granular steps which happen mostly in the background as a result of the logical replication implementation in Pglogical. Figure 13 shows the growth of data directory during the initial stages of the upgrade and highlights the individual steps there.

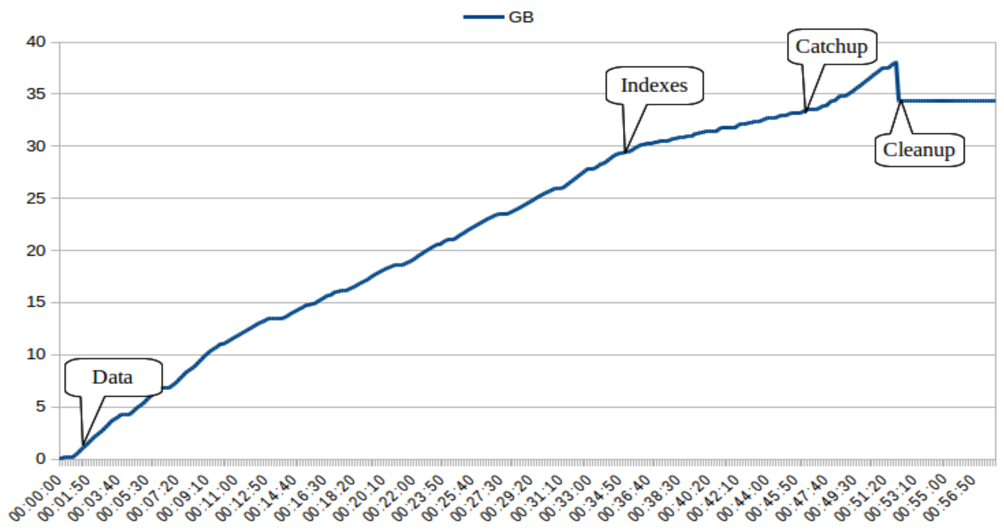

Figure 13. Graph of database size growth during logical replication initialization.

The importance of highlighting individual steps is visible in Figure 14 which shows the transaction rate and latency of requests during the same period of time.

Figure 14 shows that the data copy affects the performance of the server in a similar way the standby cloning does, decreasing transactions per second (TPS) in the benchmark also by approximately 25%. This suggests that the upgrade using pglupgrade method cannot be done on the server which does not have some spare resources during the upgrade period, so scheduling the upgrade for a period of a lower load is still important. It also shows that the actual replication, once we get to index and catch up phase have minimal effect on the server so both clusters can run in this configuration indefinitely and if needed tests can be run on the new cluster before switching the main load to it.

Another interesting point shown by the Figure 14 is the minimal downtime during the switch from the old cluster to new cluster (marked as Switch in the picture). What happens is that the TPS goes to 0 for about 3 seconds and latencies get high because the queries have to wait in the queue but once the pgbouncer moves queries to the new server, they get immediately served. In other words, the behavior proposed in the previous chapters was confirmed by the experimental run!

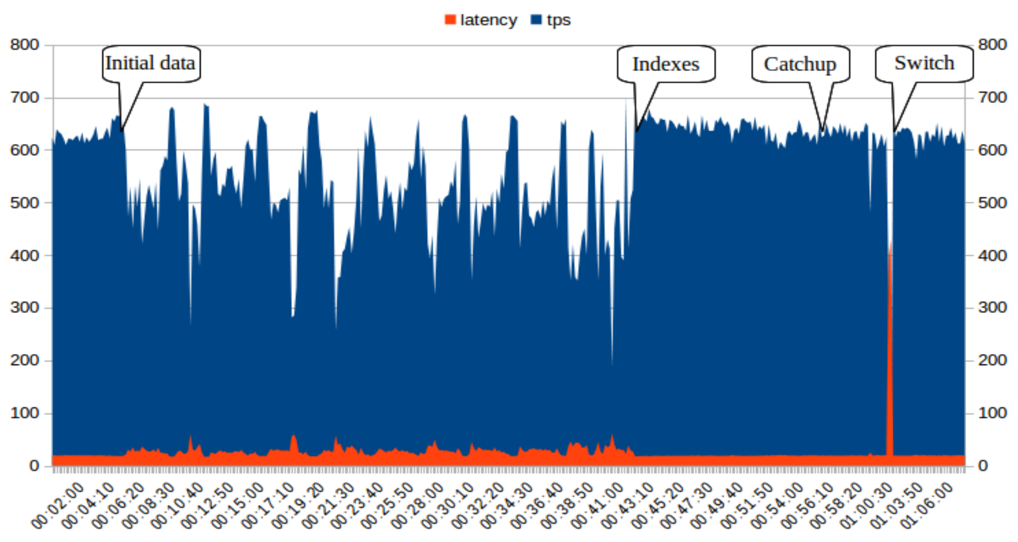

Figure 14. Transaction rate and latency graph during the upgrade process.

The figures used in this section were produced from monitoring the run of the first case (Database with 3 standby servers used for high availability). The graphs for the second case look very similar as same data size and server configuration was used for both use cases. Different data size and different server configuration would affect the time of upgrade and in the case of pg_dump/pg_restore and pg_upgrade the length of the downtime as well. The ratio of downtime between different tools would be different with different data size but using same data size was convenient to create instances easily.

6 Summary

This thesis has discussed problems associated with database upgrades with a focus on open source database management system PostgreSQL. The main problem identified in the thesis is the length of the downtime of a database cluster during the upgrade. This has a direct impact on any application which needs the database cluster being upgraded and normally means downtime or at least reduced functionality of the application as well. Based on the above, the thesis focused on a task of minimizing the downtime that is required for PostgreSQL major version upgrades and on developing a solution that also automates the whole upgrade process.

To achieve this goal, existing upgrade methods for PostgreSQL and other related databases were researched. Currently available and built-in database upgrade methods for PostgreSQL (i.e. pg_dump/pg_restore, pg_upgrade) were not feasible to accomplish near-zero downtime objective. Therefore, the author explored Logical Replication capabilities of PostgreSQL that are available for the PostgreSQL 9.4 and newer versions as the upgrade method. Primarily using the Logical Replication extension Pglogical as the base of the proposed upgrade method, the author implemented an automated PostgreSQL cluster upgrade tool, namely Pglupgrade. Pglupgrade tool also provides a graceful method for pointing applications to the new (updated) cluster by utilizing PgBouncer connection proxy tool.

Ansible configuration management and IT automation tool was used to orchestrate the upgrade process. The platform built specifically to run in Cloud to benefit from flexible nature of the Cloud Computing and pay-as-you-go billing model, as well as the ease of integration with automation tool Ansible. Pglupgrade tool provides provisioning option for Cloud instances with the required specifications include multiple Cloud regions, different instance sizes, customizable host configurations and network rules.

To evaluate the upgrade method that is developed in Pglupgrade tool, two case studies were performed. The first case study was focused on a small cluster that was set for high availability reasons. The results of the first case study were in favour of our approach comparing to the alternative upgrade methods. Pglupgrade was the sole method that did not disrupt the application and perceived as a delay by only causing a longer transaction response when only 3 seconds of downtime experienced. The second case study performed on a bigger cluster with 23 servers that were set up to scale read queries to divide system load. Pglupgrade approach outperformed other solutions by achieving minimal primary downtime of 5 seconds, without disrupting the application as in the first use-case. The results of the second experiment also proved that Pglupgrade enabled the large cluster to operate in full capacity immediately after 3 seconds.

In conclusion, this thesis suggested how database clusters can be upgraded with minimal downtime. The author has shown that by using the power of replication of logical changes and a protocol aware connection proxy it is possible to make applications oblivious to the fact that the database is being upgraded and have users of that application largely unaffected by such upgrade barring small performance drop. Using the automated configuration management and orchestration, it is also possible to make this process relatively painless and repeatable. The Pglupgrade tool itself has demonstrated the practical application of the ideas described in this thesis and proved usability of the suggested approach during the evaluation.

References

[1] P. Beynon-Davies, Database Systems, 3rd toim., Palgrave, 2003.

[2] P. Lake ja P. Crowther, „Data, an Organisational Asset," %1 *Concise Guide to Databases: A Practical Introduction,* London, : Springer London, 2013, pp. 3-19.

[3] ScaleArc, „The State of Application Uptime in Database Environments," ScaleArc, Santa Clara, CA, 2015.

[4] 2ndQuadrant, „pglogical," may 2017. Available: https://2ndquadrant.com/en/resources/pglogical/.

[5] P. Authors, „PgBouncer," may 2017. Available: https://pgbouncer.github.io/.

[6] M. Attaran ja S. Attaran, „Collaborative supply chain management: The most promising practice for building efficient and sustainable supply chains," *Business Process Management Journal,* kd. 13, pp. 390-404, 2007.

[7] M. Attaran, „Information technology and business- process redesign," *Business Process Management Journal,* kd. 9, pp. 440-458, 2003.

[8] A. Moenkeberg, P. Zabback, C. Hasse ja G. Weikum, „The COMFORT Prototype: A Step Towards Automated Database Performance Tuning," *SIGMOD Rec.,* kd. 22, pp. 542-543, #jun# 1993.

[9] C. Deba , E. Kurt, A. Ashish ja A. Waleed, „Manageability with Oracle Database 12c," Oracle Corporation, Redwood Shores, CA 94065, 2013.

[10] C. M. Garcia-Arellano, S. S. Lightstone, G. M. Lohman, V. Markl ja A. J. Storm, „Autonomic features of the IBM DB2 universal database for linux, UNIX, and windows," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews),* kd. 36, pp. 365-376, May 2006.

[11] S. Agrawal, N. Bruno, S. Chaudhuri ja V. Narasayya, „AutoAdmin: Self-Tuning Database Systems Technology," %1 *Data Engineering Bulletin,* 2006.

[12] S. Chaudhuri, A. C. Konig ja V. Narasayya, „SQLCM: a continuous monitoring framework for relational database engines," %1 *Proceedings. 20th International Conference on Data Engineering,* 2004.

[13] R. S. Xin, W. McLaren, P. Dantressangle, S. Schormann, S. Lightstone ja M. Schwenger, „MEET DB2: automated database migration evaluation," *Proceedings of the VLDB Endowment,* kd. 3, pp. 1426-1434, 2010.

[14] I. B. M. Redbooks, Migrating from Oracle . . . to IBM Informix Dynamic Server on Linux, Unix, and Windows, Vervante, 2009.

[15] Oracle, Oracle Migration Workbench User's Guide, Redwood Shores, CA 94065: Oracle, 2005.

[16] S. Gajre, B. Bordia ja V. Soni, „Migration to Microsoft SQL Server 2014 Using SSMA," Microsoft Corporation, Albuquerque, New Mexico, 2015.

[17] S. Abdul Khalek ja S. Khurshid, „Automated SQL Query Generation for Systematic Testing of Database Engines,“ %1 *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* New York, NY, USA, 2010.

[18] K. Taneja, Y. Zhang ja T. Xie, „MODA: Automated Test Generation for Database Applications via Mock Objects,“ %1 *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* New York, NY, USA, 2010.

[19] R. Nehme ja N. Bruno, „Automated Partitioning Design in Parallel Database Systems,“ %1 *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data,* New York, NY, USA, 2011.

[20] C. Curino, H. J. Moon ja C. Zaniolo, *Automating Database Schema Evolution in Information System Upgrades.*

[21] I. Samoladas, I. Stamelos, L. Angelis ja A. Oikonomou, „Open Source Software Development Should Strive for Even Greater Code Maintainability,“ *Commun. ACM,* kd. 47, pp. 83-87, #oct# 2004.

[22] G. D. G. PostgreSQL, „PostgreSQL Case Studies,“ apr 2017. [Võrgumaterjal]. Available: https://www.postgresql.org/about/casestudies/.

[23] I. Red Hat, „Ansible Case Studies,“ apr 2017. [Võrgumaterjal]. Available: https://www.ansible.com/case-studies.

[24] S. Walli, D. Gynn ja B. V. Rotz, „The Growth of Open Source Software in Organizations,“ Optaros, Inc., Boston, MA, 2005.

[25] UBM, „Linux Outlook,“ InformationWeek, San Francisco, CA, 2005.

[26] H. a. V. W. a. W. J. Chesbrough, Open Innovation: Researching a New Paradigm, 2006.

[27] A. Metiu ja B. Kogut, *Distributed Knowledge and the Global Organization of Software Development,* 2002.

[28] E. F. Codd, „A Relational Model of Data for Large Shared Data Banks,“ *Commun. ACM,* kd. 13, pp. 377-387, #jun# 1970.

[29] iDatalabs, „Market Share of Database Management System products,“ apr 2017. Available: https://idatalabs.com/tech/database-management-system.

[30] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes ja R. E. Gruber, „Bigtable: A Distributed Storage System for Structured Data,“ *ACM Trans. Comput. Syst.,* kd. 26, pp. 4q1--4q26, #jun# 2008.

[31] A. a. M. J. Lith, „Investigating Storage Solutions for Large Data,“ 2010.

[32] D. Pritchett, „BASE: An Acid Alternative,“ *Queue,* kd. 6, pp. 48-55, #may# 2008.

[33] S. Gilbert ja N. Lynch, „Brewerś Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services,“ *SIGACT News,* kd. 33, pp. 51-59, #jun# 2002.

[34] P. Warden, Big Data Glossary: A Guide to the New Generation of Data Tools, OŔeilly Media, 2011.

[35] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem ja P. Helland, „The End of an Architectural Era: (Itś Time for a Complete Rewrite)," %1 *Proceedings of the 33rd International Conference on Very Large Data Bases,* Vienna, 2007.

[36] Oracle, „Oracle Database Rolling Upgrades Using a Data Guard Physical Standby Database," 2016.

[37] M. J. Kavis, Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS) (Wiley CIO), Wiley, 2014.

[38] P. Mell ja T. Grance, „The NIST definition of cloud computing," *National Institute of Standards and Technology,* kd. 53, p. 50, 2009.

[39] A. Wittig ja M. Wittig, Amazon Web Services in Action, Manning Publications, 2015.

[40] D. A. Zapolsky, „2015 Letter to Shareholders," 2016.

[41] L. Leong, P. Gregor, B. Gill ja M. Dorosh, „Magic Quadrant for Cloud Infrastructure as a Service, Worldwide," 2016.

[42] F. A. Locati, Learning Ansible 2 - Second Edition, Packt Publishing - ebooks Account, 2016.

[43] G. Yıldırım, „Pglupgrade repository," may 2017. Available: https://gitlab.com/gulcin/pglupgrade.

[44] G. Yıldırım, „Managing PostgreSQL with Ansible PGConf.EU," 2015. Available: http://slides.com/apatheticmagpie/managing-postgres-withansible#/.

[45] G. Yıldırım, „Managing PostgreSQL with Ansible FOSDEM PGDay," 2016. Available: http://slides.com/apatheticmagpie/managing-postgres-with-ansiblefosdem#/.

[46] G. Yıldırım, „Ansible Loves PostgreSQL 5432...MeetUs! Milano," 2016. Available: http://slides.com/apatheticmagpie/ansible-loves-postgres#/.

[47] G. Yıldırım, „Ansible Loves PostgreSQL," apr 2017. Available: https://blog.2ndquadrant.com/ansible-loves-postgresql/.

[48] G. Yıldırım, „PostgreSQL Planet in Ansible Galaxy," apr 2017. Available: https://blog.2ndquadrant.com/postgresql-planet-in-ansible-galaxy/.

[49] P. Jelinek, „Logical Replication in PostgreSQL," 2016.

Appendix 1 - Pglupgrade Playbook

Pglupgrade.yml

```
- name: Build hosts based on configuration
 hosts: new-standbys
 vars_files:
     - config.yml
 tasks:
   - add_host: name={{item}} group=new-initial-standbys
     with_items: "{{groups['new-standbys'][:initial_standbys]}}"
   - add_host: name={{item}} group=new-other-standbys
     with_items: "{{groups['new-standbys'][initial_standbys:]}}"

- name: Setup new cluster with {{initial_standbys}} standby(s)
 hosts: new-primary, new-initial-standbys
 become: true
 become_user: postgres
 vars_files:
     - config.yml
 roles:
     - role: postgres/remove
     - role: postgres/pkg
     - role: postgres/primary
       when: inventory_hostname in groups['new-primary']
     - role: postgres/standby
       when: inventory_hostname in groups['new-standbys']

- name: Modify the old primary to support logical replication
 hosts: old-primary
 become: true
 become_user: postgres
 vars_files:
     - config.yml
 roles:
     - role: pglogical/common
     - role: pglogical/publisher

- name: Replicate to the new primary
 hosts: new-primary
 become: true
 become_user: postgres
 vars_files:
     - config.yml
 roles:
     - role: pglogical/common
     - role: pglogical/subscriber

- name: Switch the pgbouncer (and applications) to new primary
 hosts: pgbouncer
 become: true
 become_user: postgres
 vars_files:
     - config.yml
 roles:
     - role: pgbouncer
       tags:
         - pgbouncer

- name: Clean up the replication setup between old primary and new primary
 hosts: new-primary, old-primary
 serial: 1
 become: true
 become_user: postgres
 vars_files:
     - config.yml
 roles:
     - role: pglogical/cleanup
       tags:
         - cleanup

- name: Stop the old cluster
 hosts: old-primary, old-standbys
 become: true
 become_user: root
 vars_files:
     - config.yml
 tasks:
     - service:
         name: postgresql
         state: stopped
       tags:
         - cleanup

- name: Reconfigure rest of the standbys for the new primary
 hosts: new-other-standbys
 become: true
 become_user: postgres
 vars_files:
     - config.yml
 roles:
     - role: postgres/remove
     - role: postgres/pkg
     - role: postgres/standby
```

Appendix 2 - Postgres Roles

Postgres/Conf Role
postgres/conf/tasks/main.yaml

```
- block:
 - name: Ensure the config directory exists
   file:
     path: "{{ postgres_new_confdir }}"
     state: directory
     owner: postgres
     group: postgres
     mode: 0700

 - name: Install new postgresql.conf
   template:
     src: postgres/conf/templates/postgresql.conf.j2
     dest: "{{ postgres_new_confdir }}/postgresql.conf"
     owner: postgres
     group: postgres
     mode: 0600

 - name: Install new pg_hba.conf
   template:
     src: postgres/conf/templates/pg_hba.conf.j2
     dest: "{{ postgres_new_confdir }}/pg_hba.conf"
     owner: postgres
     group: postgres
     mode: 0600

 become: yes
 become_user: root
```

postgres/conf/templates/pg_hba.conf.j2

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     peer
host    all             all             0.0.0.0/0               trust
local   replication     all                                     peer
host    replication     all             0.0.0.0/0               trust
```

postgres/conf/templates/postgresql.conf.j2

```
data_directory = '{{ postgres_new_datadir }}'
hba_file = '{{ postgres_new_confdir }}/pg_hba.conf'

listen_addresses = '*'
max_connections = 100
hot_standby = on
```

```
wal_level = 'logical'
max_worker_processes = 50
max_replication_slots = 50
max_wal_senders = 50
shared_preload_libraries = 'pglogical'
```

Postgres/Pkg Role
postgres/pkg/tasks/main.yml

```
- name: Install Postgres packages
 become: yes
 become_user: root
 package: name={{ item }} state=latest
 with_items: "{{ postgres_packages }}"
```

postgres/pkg/defaults/main.yml

```
postgres_packages:
   - postgresql-{{postgres_new_version}}
   - postgresql-{{postgres_new_version}}-dbg
   - postgresql-client-{{postgres_new_version}}
   - postgresql-contrib-{{postgres_new_version}}
   - postgresql-server-dev-{{postgres_new_version}}
   - postgresql-plperl-{{postgres_new_version}}
   - postgresql-{{postgres_new_version}}-plv8
   - postgresql-{{postgres_new_version}}-pglogical
```

Postgres/Primary Role
postgres/primary/tasks/main.yml

```
- name: Check to see if the data directory is empty
 stat: path="{{ postgres_new_datadir }}/PG_VERSION"
 register: pgdata
 tags: [postgres, initdb]

- name: Create new data directory
 become: yes
 become_user: root
 command: "pg_createcluster {{ postgres_new_version }} main"
 when: not pgdata.stat.exists

- include: postgres/conf/tasks/main.yaml

- name: Start Postgres
 become: yes
 become_user: root
 service:
   name: postgresql
   state: restarted
- name: Ensure pglupgrade user exists
```

```
 postgresql_user:
     name: "{{ replica_user }}"
     password: "{{ replica_pass }}"
     encrypted: true
     role_attr_flags: REPLICATION,LOGIN
     state: present

- name: Ensure the database exists on the new server
 postgresql_db:
     db: "{{ pglupgrade_database }}"
```

Postgres/Standby Role
postgres/standby/tasks/main.yml

```
- name: Stop Postgres if necessary
 become: yes
 become_user: root
 service:
   name: postgresql
   state: stopped

- name: Ensure the data directory is empty
 become: yes
 become_user: root
 file:
   path: "{{ postgres_new_datadir }}"
   state: absent

- name: Ensure the data directory exists
 become: yes
 become_user: root
 file:
   path: "{{ postgres_new_datadir }}"
   state: directory
   owner: postgres
   group: postgres
   mode: 0700

- name: Create clone of the master
 command: pg_basebackup -w -c fast -X stream -R -d "{{ postgres_new_dsn }}" -
U "{{ replica_user }}" -D "{{ postgres_new_datadir }}"

- include: postgres/conf/tasks/main.yaml

- name: Start Postgres
 become: yes
 become_user: root
 service:
   name: postgresql@9.6-main
   state: started
```

Postgres/Remove Role
postgres/remove/main.yml

```
- name: Ensure that Postgres is not running
 service: name=postgresql state=stopped

- name: Ensure the old config and data directories don't exist
 file:
   path: "{{ item }}"
   state: absent
 with_items:
   - "{{ postgres_old_datadir }}"
   - "{{ postgres_old_confdir }}"
```

Appendix 3 - Pglogical Roles

Pglogical/Common Role
pglogical/common/tasks/main.yml

```
- name: Ensure pglupgrade user exists
 postgresql_user:
     name: "{{ pglupgrade_user }}"
     password: "{{ pglupgrade_pass }}"
     encrypted: true
     role_attr_flags: SUPERUSER,REPLICATION,LOGIN
     state: present

- name: Ensure pglogical extension exists in user databases
 postgresql_ext:
     db: "{{ pglupgrade_database }}"
     name: pglogical
```

Pglogical/Publisher Role
pglogical/publisher/tasks/main.yaml

```
- name: Create node
 command: psql -qAtw -c "SELECT pglogical.create_node('{{subscription_name}}-
old', '{{ postgres_old_dsn }}')"

- name: Create replication set
 command: psql -qAtw -c "SELECT
pglogical.create_replication_set('{{replication_set}}')"

- name: Gather tables
 command: psql -qAtw -c "SELECT oid FROM pg_catalog.pg_class WHERE relkind =
'r' AND relpersistence = 'p' AND oid >= 16384 AND NOT relnamespace =
ANY(ARRAY(SELECT oid FROM pg_catalog.pg_namespace WHERE oid = 11 OR nspname
IN ('pglogical', 'information_schema')))"
 register: tables

- name: Add tables to replication set
 command: psql -qAtw -c "SELECT
pglogical.replication_set_add_table('{{replication_set}}', '{{item}}',
false);"
 with_items: "{{ tables.stdout_lines }}"

- name: Gather sequences
 command: psql -qAtw -c "SELECT oid FROM pg_catalog.pg_class WHERE relkind =
'S' AND relpersistence = 'p' AND oid >= 16384 AND NOT relnamespace =
ANY(ARRAY(SELECT oid FROM pg_catalog.pg_namespace WHERE oid = 11 OR nspname
IN ('pglogical', 'information_schema')))"
 register: sequences

- name: Add sequences to replication set
```

```
 command: psql -qAtw -c "SELECT
pglogical.replication_set_add_sequence('{{replication_set}}', '{{item}}',
false);"
 with_items: "{{ sequences.stdout_lines }}"
```

Pglogical/Subscriber Role

pglogical/subscriber/tasks/main.yaml

```
- name: Create node
 command: "psql -qAtw -c \"SELECT
pglogical.create_node('{{subscription_name}}-new', '{{ postgres_new_dsn
}}')\""

- name: Create the subscription
 command: "psql -qAtw -c \"SELECT
pglogical.create_subscription('{{subscription_name}}', '{{ postgres_old_dsn
}}', ARRAY['{{ replication_set }}'], synchronize_structure := true,
synchronize_data := true)\""

- name: Wait for subscription to be ready (this will take a while)
 command: "psql -qAtw -c \"SELECT status FROM
pglogical.show_subscription_status('{{subscription_name}}')\""
 register: result
 until: result.stdout.find("replicating") != -1
 delay: 10
 retries: "{{sync_wait_time|default(10000)}}"
```

Pglogical/Cleanup Role

pglogical/cleanup/tasks/main.yml

```
- name: Drop subscription if preset
 command: psql -qAtw -c "SELECT pglogical.drop_subscription('upgrade',
true);" -d "{{pglupgrade_database}}"

- name: Drop local node if preset
 command: psql -qAtw -c "SELECT pglogical.drop_node(sub_name, true) FROM
pglogical.subscription WHERE sub_name IN ('{{subscription_name}}-old',
'{{subscription_name}}-new');" -d "{{pglupgrade_database}}"

- name: Remove pglogical extension
 command: psql -qAtw -c "DROP EXTENSION IF EXISTS pglogical CASCADE" -d
"{{pglupgrade_database}}"

- name: Remove the upgrade user
 postgresql_user:
     name: "{{ pglupgrade_user }}"
     state: absent
```

Appendix 4 - PgBouncer Role

pgbouncer/tasks/main.yaml

```
- set_fact:
   pgbouncer_dsn: "port=6432 dbname=pgbouncer user={{pgbouncer_user}}"

- name: Wait for subscription to catch up
 # Wait to get less than 1 WAL file behind so that the following pause is
short
 command: "psql -qAtw -d \"{{ postgres_old_dsn }}\" -c \"SELECT 'ok' FROM
pg_catalog.pg_stat_replication s WHERE s.application_name = 'upgrade' AND
pg_xlog_location_diff(pg_current_xlog_location(), write_location) <
16000000\""
 register: result
 until: result.stdout.find("ok") != -1
 delay: 10
 retries: "{{sync_wait_time|default(1000)}}"

- name: Pause pgbouncer
 command: "psql -qAtw -d \"{{ pgbouncer_dsn }}\" -c \"PAUSE;\""

- name: Update pgbouncer config
 become: yes
 become_user: root
 replace:
     path: /etc/pgbouncer/pgbouncer.ini
     regexp: "{{ groups['old-primary'][0]|regex_escape() }}"
     replace: "{{ groups['new-primary'][0] }}"
     backup: true
     owner: postgres
     group: postgres
     mode: 0640

- name: Reload pgbouncer config
 command: "psql -qAtw -d \"{{ pgbouncer_dsn }}\" -c \"RELOAD;\""

- name: Wait for replication to fully catch up
 command: "psql -qAtw -d \"{{ postgres_old_dsn }}\" -c \"SELECT 'ok' FROM
pg_catalog.pg_stat_replication s WHERE s.application_name = 'upgrade' AND
s.flush_location >= pg_current_xlog_location()\""
 register: result
 until: result.stdout.find("ok") != -1
 delay: 10
 retries: 10
 ignore_errors: yes

- name: Resume pgbouncer
 command: "psql -qAtw -d \"{{ pgbouncer_dsn }}\" -c \"RESUME;\""
```

Appendix 5 - Provision Playbook

Provision.yml

```
- name: Provision servers
 hosts: 127.0.0.1 # localhost
 vars_files:
     - config.yml
     - config-aws.yml

 roles:
     - role: aws/provision
```

config-aws.yml

```
ec2_ami_name: "ubuntu/images/ebs/ubuntu-trusty-16.04-amd64-server-*"
ec2_ami_owner: 099720109477
ec2_ssh_user: ubuntu
ec2_ssh_key: ~/.ssh/id_rsa.pub
ec2_ssh_key_name: postgresql-key
ec2_vpc_name: Test

ec2_regions:
 eu-west-1:
     subnet: 10.33.0.0/16
 eu-central-1:
     subnet: 10.33.0.0/16

servers:
   - role: old-master
     type: t2.micro
     region: eu-west-1
     volume_size: 50
     count: 1
     roles:
   - role: new-master
     type: t2.micro
     region: eu-west-1
     volume_size: 50
     count: 1
   - role: pgbouncer
     type: t2.micro
     region: eu-west-1
     volume_size: 50
     count: 1
   - role: standby
     type: t2.micro
     region: eu-west-1
     volume_size: 50
```

```
  count: 1
- role: standby
  type: t2.micro
  region: eu-central-1
  volume_size: 50
  count: 1
```

Appendix 6 - AWS/Provision Role

aws/provision/tasks/main.yml

```
- name: Ensure the SSH key is present
 ec2_key:
   state: present
   region: "{{ item.key }}"
   name: "{{ ec2_ssh_key_name }}"
   key_material: "{{ lookup('file', ec2_ssh_key) }}"
 with_dict: "{{ ec2_regions }}"

- name: Configure VPCs
 include: 'vpc.yml'
 with_dict: "{{ ec2_regions }}"
 loop_control:
   loop_var: region

- name: Configure AMIs
 include: 'ami.yml'
 with_dict: "{{ ec2_regions }}"
 loop_control:
   loop_var: region

- name: Ensure master EC2 instances & volumes are present
 ec2:
   assign_public_ip: yes # our machines should access internet
   instance_tags: { name: "pgl-{{ item.role }}-{{ item.region-item.0+1 }}",
role: "{{ item.role }}", region: "{{ item.region }}" }
   exact_count: "{{ item.count }}"
   count_tag:
     role: "{{ item.role }}"
     region: "{{ item.region }}"
   image: "{{ ec2_regions[item.region].ami_id }}"
   instance_type: "{{ item.type }}"
   group_id: "{{ ec2_regions[item.region].security_group_id }}"
   key_name: "{{ ec2_ssh_key_name }}"
   region: "{{ item.region }}"
   volumes:
     - device_name: /dev/sdc
       volume_size: "{{ item.volume_size }}"
       delete_on_termination: false
   vpc_subnet_id: "{{ ec2_regions[item.region].subnet_id }}"
   wait: yes
 register: ec2
 with_items: "{{ servers }}"

- name: Wait for SSH to become ready
 wait_for:
     host: "{{ item.public_ip }}"
```

```
        port: 22
        timeout: 320
        state: started
    with_items: "{{ ec2.instances }}"
```

aws/provision/tasks/vpc.yml

```
- name: Ensure VPC is present
  ec2_vpc_net:
    state: present
    name: "{{ ec2_vpc_name }}"
    region: "{{ region.key }}"
    cidr_block: "{{ region.value.subnet }}"
  register: vpc

- name: "Register vpc {{ vpc.vpc.id }} region {{ region.key }}"
  set_fact:
    ec2_regions: "{{
      ec2_regions|default({})|combine({
          region.key: {'vpc_id': vpc.vpc.id}
      }, recursive=True)
    }}"

- name: Create internet gateway for VPC
  ec2_vpc_igw:
    vpc_id: "{{ vpc.vpc.id }}"
    region: "{{ region.key }}"
    state: present

- name: Create subnets
  ec2_vpc_subnet:
    state: present
    cidr: "{{ region.value.subnet }}"
    vpc_id: "{{ vpc.vpc.id }}"
    region: "{{ region.key }}"
  register: subnet

- name: "Register subnet {{ subnet.subnet.id }} region {{ region.key }}"
  set_fact:
    ec2_regions: "{{
      ec2_regions|default({})|combine({
          region.key: {'subnet_id': subnet.subnet.id}
      }, recursive=True)
    }}"

- name: Create VPC route table
  ec2_vpc_route_table:
    region: "{{ region.key }}"
    vpc_id: "{{ vpc.vpc.id }}"
    subnets:
      - "{{ region.value.subnet }}"
    routes:
      - dest: 0.0.0.0/0
        gateway_id: igw

- name: Ensure the PostgreSQL security group is present
  ec2_group:
    state: present
    vpc_id: "{{ vpc.vpc.id }}"
    region: "{{ region.key }}"
    name: "{{ ec2_vpc_name }} - pglPostgreSQL"
    description: "Security group for PostgreSQL database servers"
    rules:
      - proto: tcp
        from_port: 22
        to_port: 22
        cidr_ip: 0.0.0.0/0
      - proto: tcp
        from_port: 5432
        to_port: 5432
        cidr_ip: 0.0.0.0/0
      - proto: all
        from_port: -1
        to_port: -1
        cidr_ip: "{{ region.value.subnet }}"
  register: security_group

- name: "Register security group {{ security_group.group_id }} region {{
region.key }}"
  set_fact:
    ec2_regions: "{{
      ec2_regions|default({})|combine({
          region.key: {'security_group_id': security_group.group_id}
      }, recursive=True)
    }}"
```

aws/provision/tasks/ami.yml

```
- name: Find the ami
  ec2_ami_find:
    name: ec2_ami_name
    owner: ec2_ami_owner
    region: "{{ region.key }}"
    sort: name
    sort_order: descending
    sort_end: 1
  register: ami_find

- name: "Register ami {{ ami_find.results[0].ami_id }} region {{ region }}"
  set_fact:
    ec2_regions: "{{
```

```
  ec2_regions|default({})|combine({
      region.key: {'ami_id': ami_find.results[0].ami_id}
  }, recursive=True)
}}"
```