UNITED NATIONS

unite

Project UN-OSM Database Platform

END PROJECT REPORT

Revision History

Date	Version	Description	Author
Select a date.	0.1	First Draft	Type here.
Select a date.	Type here.	Type here.	Type here.
Select a date.	Type here.	Type here.	Type here.
Select a date.	Type here.	Type here.	Type here.
Select a date.	Type here.	Type here.	Type here.

Copyright

The present document, including any of its contents, cannot be copied or reproduced in any form without prior approval of the Office of Information and Communications Technology of the United Nations.

Confidentiality

This Document is classified as Confidential (as per ST/SGB/2007/6) as it includes information on the status of systems of the Office of Information and Communications Technology of the United Nations and should be treated with care to ensure only trusted parties have access to the information contained herein.

Table of Contents

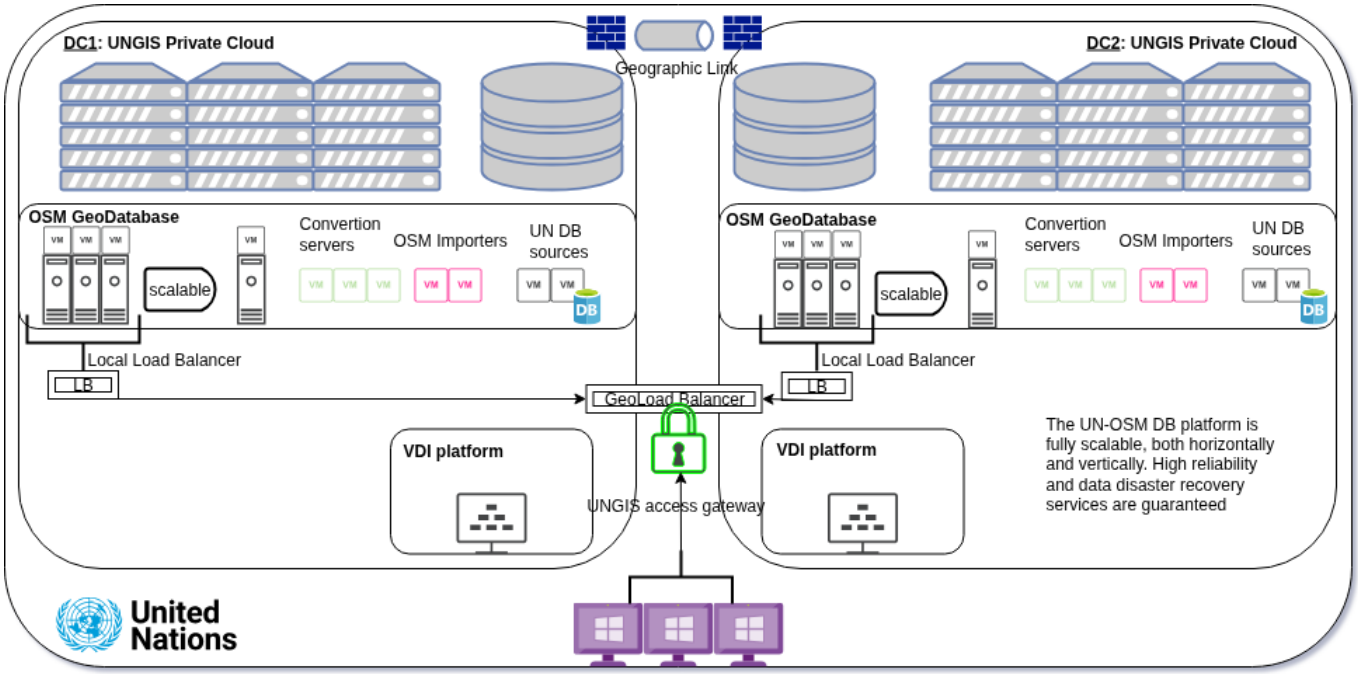
1. Introduction

This document offers a comprehensive technical overview and detailed step-by-step instructions for setting up and managing a PostgreSQL 15 cluster with asynchronous replication across multiple nodes. It outlines the configuration processes for both master and slave nodes, ensuring effective administration of the platform.

This guide is designed for database administrators (DBAs) and system administrators who will be managing or working with the UN-OSM platform. Key topics covered in this document include installation, configuration, replication setup, troubleshooting, performance tuning, backup strategies, and best practices for administering the UN-OSM platform.

2. System Architecture

High-level architecture of the UN-OSM PostgreSQL platforms showing three nodes, one master and one slave node in Brindisi and one slave node in Valencia.



The core database layer is built on four dedicated servers, unosm-r-db-01 through unosm-r-db-03, configured for asynchronous replication to ensure scalability and fault tolerance. The master node serves as the primary server, handling all write operations—INSERT, UPDATE, DELETE—managing transactions and data processing locally. Changes are recorded in Write-Ahead Logs (WAL), which are asynchronously replicated to slave nodes located in Brindisi and Valencia, acting as read-only replicas. This replication method improves write performance by allowing the master to commit transactions without waiting for slaves to acknowledge receipt of WAL logs. However, asynchronous replication can introduce some delay, meaning the slaves may not reflect changes in real time.

Data ingestion is managed by the unosm-r-app-01 server, which imports map data using the imposm import tool. Client requests are load balanced on the database slave nodes unosm-r-db-02 and unosm-r-db-03 across two servers: unosm-r-app-02 at Brindisi and unosm-p-app-02 in Valencia, although active load balancing currently happens only within the Brindisi site. These application servers distribute user queries efficiently across the database nodes to optimize performance and reliability. This load balancing is overseen by a NetScaler load balancer set up geographically, directing traffic to the osmpgsql.dfs.un.org endpoint. It routes user requests intelligently based on factors like client location, site workload, and performance metrics, thereby reducing latency and balancing infrastructure use between the Brindisi and Valencia data centers.

The asynchronous slave nodes offload read queries, reducing the master's workload while providing redundancy and enhancing disaster recovery capabilities due to their geographic distribution. However, in the event of master failure,

manual promotion of a slave to master status is required, as automatic failover is not implemented. Additionally, the unosm-r-data-01 server is dedicated to export tasks, aiding data dissemination when needed.

Overall, this multi-layered, geographically distributed architecture reflects a strategic design focused on delivering reliable, scalable, and evolving geospatial services. It prioritizes read scalability and fault tolerance, underpinning critical mapping initiatives essential for humanitarian and operational efforts worldwide.

The table below presents the host names along with their corresponding virtual hardware features:

HOST-NAME	ROLE	CPU	RAM (GB)	DISK (GB)
unosm-r-app-02	Master Brindisi	4	8	100
unosm-p-app-02	Slave Brindisi	4	8	100

Table 1: Ha-Proxy Local Load Balancer Servers

HOST-NAME	ROLE	CPU	RAM (GB)	DISK (TB)
unosm-r-db-01	Master Brindisi	16	192	1.5
unosm-r-db-02	Slave Brindisi	16	192	1.5
unosm-r-db-03	Slave Brindisi	16	192	1.5

Table 2: OSM PostgreSQL Database Servers

MASTER NODE DNS ALIAS	osmpgsql.dfs.un.org

3. PostgreSQL Installation and Configuration

This paragraph offers a concise overview of the steps involved in setting up PostgreSQL on Red Hat 9 Linux. The installation process begins with ensuring that your system meets the necessary prerequisites, such as having the appropriate repositories enabled. Following this, users can install PostgreSQL using the package manager, which simplifies the installation of all required dependencies. Once installed, initial configuration tasks include setting up the PostgreSQL service to start automatically and configure it.

This overview serves as a foundation for a more detailed guide on effectively deploying PostgreSQL in a Red Hat 9 environment.

3.1 Installation steps for PostgreSQL 15

To install PostgreSQL 15 on Red Hat 9, follow these steps:

- Enable the PostgreSQL Repository: as “root” user, first, add the PostgreSQL repository to your system:dnf install install https://download.postgresql.org/pub/repos/yum/reporpms/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm
- Install PostgreSQL 15: once the repository is added, install PostgreSQL 15 with its environment:

  dnf install -y postgresql15-server postgresql15-llvmjit postgresql15-contrib postgis33_15-utils postgis33_15
- Initialize the Database: initialize the PostgreSQL database:

  /usr/pgsql-15/bin/postgresql-15-setup initdb
- Enable and Start PostgreSQL Service: Enable PostgreSQL to start on boot and then start the service:

  systemctl enable --now postgresql-15
- Verify Installation: Check the PostgreSQL version to confirm the installation:

  psql --version

3.2 Configuration Files Overview

In PostgreSQL 15, configuration files play a crucial role in defining the database server's behavior and security settings. The two primary configuration files are postgresql.conf and pg_hba.conf. The postgresql.conf file contains settings for server performance, memory usage, logging, and connection parameters, while the pg_hba.conf file manages client authentication, specifying which users can connect from which hosts and using which authentication methods. Proper configuration of these files ensures optimal performance and secure access to the PostgreSQL database.

postgresql.conf:

```
listen_addresses = '*'                  # what IP address(es) to listen on;
port = 5432                             # (change requires restart)
max_connections = 60                    # (change requires restart)
shared_buffers = 8GB                    # min 128kB
temp_buffers = 512MB                    # min 800kB
work_mem = 34952kB                      # min 64kB
maintenance_work_mem = 2GB              # min 1MB
effective_cache_size = 24GB
huge_pages = on                         # on, off, or try
effective_io_concurrency = 200          # 1-1000; 0 disables prefetching
max_stack_depth = 4MB                   # min 100kB
wal_compression = on                    # enable compression of full-page writes
max_worker_processes = 8                # (change requires restart)
max_parallel_workers_per_gather = 4     # taken from max_parallel_workers
max_parallel_workers = 8                # maximum number of max_worker_processes that
max_parallel_maintenance_workers = 4    # taken from max_parallel_workers
max_locks_per_transaction = 2000        # min 10
max_files_per_process = 2000            # min 64
force_parallel_mode = on
parallel_setup_cost = 400               # same scale as above
parallel_tuple_cost = 0.04              # same scale as above
enable_partitionwise_join = on
enable_partitionwise_aggregate = on
constraint_exclusion = on               # on, off, or partition
enable_partition_pruning = on
seq_page_cost = 5                       # measured on an arbitrary scale
random_page_cost = 1.1                  # same scale as above
checkpoint_timeout = 5min               # range 30s-1d
checkpoint_completion_target = 0.9      # checkpoint target duration, 0.0 - 1.0
checkpoint_flush_after = 512kB          # measured in pages, 0 disables
wal_writer_flush_after = 1MB            # measured in pages, 0 disables
synchronous_commit = on                 # synchronization level;
enable_seqscan = on
cpu_tuple_cost = 0.01                   # same scale as above
cpu_index_tuple_cost = 0.005            # same scale as above
cpu_operator_cost = 0.0025              # same scale as above
jit_provider = 'llvmjit'                # JIT library to use
jit = on                                # allow JIT compilation
jit_above_cost = 10000                  # perform JIT compilation if available
```

```
jit_inline_above_cost = 50000          # inline small functions if query is
jit_optimize_above_cost = 50000        # use expensive JIT optimizations if
shared_preload_libraries = 'pg_stat_statements' # (change requires restart)
pg_stat_statements.max = 10000
pg_stat_statements.track = all
default_statistics_target = 100        # range 1-10000
track_activities = on
track_counts = on
track_activity_query_size = 2048       # (change requires restart)
archive_mode = on                      # enables archiving; off, on, or always
archive_command = '/bin/true'          # command to use to archive a logfile segment
wal_level = replica                    # minimal, replica, or logical
wal_log_hints = on                     # also do full page writes of non-critical updates
max_wal_senders = 10                   # max number of walsender processes
wal_keep_size = 1536                   # wal_keep_segments * wal_segment_size (typically 16MB),
wal_keep_segments was setted to 96
max_replication_slots = 10             # max number of replication slots
max_slot_wal_keep_size = 1024MB        # in megabytes; -1 disables
hot_standby = on                       # "off" disallows queries during recovery
wal_buffers = 16MB                     # min 32kB, -1 sets based on shared_buffers
min_wal_size = 1GB
max_wal_size = 4GB
log_connections = on
log_disconnections = on
log_checkpoints = on
log_min_duration_statement = 10000     # -1 is disabled, 0 logs all statements
log_lock_waits = on                    # log lock waits >= deadlock_timeout
log_temp_files = 0                     # log temporary files equal or larger
autovacuum = on                        # Enable autovacuum subprocess?  'on'
autovacuum_analyze_scale_factor = 0.025 # fraction of table size before analyze
autovacuum_analyze_threshold = 5000    # min number of row updates before
autovacuum_max_workers = 9             # max number of autovacuum subprocesses
autovacuum_vacuum_scale_factor = 0.05  # fraction of table size before vacuum
autovacuum_vacuum_threshold = 5000     # min number of row updates before
max_standby_streaming_delay = 900s     # max delay before canceling queries
log_autovacuum_min_duration = 0        # -1 disables, 0 logs all actions and
tcp_keepalives_idle=120
tcp_keepalives_interval=20
tcp_keepalives_count=6
client_connection_check_interval=5000  # 5 second interval in milliseconds
log_line_prefix = '%m [%p] %q%u@%d (%r) '  # special values:
```

The PostgreSQL 15 configuration parameters listed in the table above, control various aspects of server behavior, performance, and logging. `listen_addresses = '*'` specifies which IP addresses PostgreSQL should listen to (* mean each address), and `port = 5432` defines the default connection port. The parameter `max_connections = 60` limits the number of concurrent connections, while `shared_buffers = 8GB` allocates memory for caching data. The parameter `temp_buffers = 512MB` and `work_mem = 34952kB` manage memory for temporary operations and query sorting, respectively.

The parameter `maintenance_work_mem = 2GB` optimizes maintenance tasks like VACUUM, while `effective_cache_size = 24GB` helps the planner choose between indexes and sequential scans. Parallelism settings like `max_parallel_workers_per_gather = 4` and `max_parallel_workers = 8` improve query performance by enabling parallel query execution. Replication settings like `wal_level = replica`, `archive_mode = on`, and `max_wal_senders = 10` ensure data durability and availability in standby servers, while `hot_standby = on` allows query execution on recovery standbys. Autovacuum settings (`autovacuum = on`, `autovacuum_analyze_scale_factor = 0.025`, etc.) automate maintenance tasks, reclaiming storage and keeping tables up-to-date. Logging options like `log_connections = on`, `log_checkpoints = on`, and `log_min_duration_statement = 10000` help monitor connections, checkpoints, and slow queries. `jit = on` enables Just-In-Time compilation for query optimization, while `max_replication_slots = 10` defines replication slot limits. TCP keep-alive settings (`tcp_keepalives_idle = 120`, `tcp_keepalives_interval = 20`, and `tcp_keepalives_count = 6`) help maintain stable client connections (this to solve the issues we have on the main routers). Lastly, `log_line_prefix` defines the format for log entries, providing detailed information on each log event. These parameters should be adjusted according to system resources, workload, and performance goals to ensure optimal operation.

- **pg_hba.conf**: is used to configure the client authentication and allow replication connections.

4. Replication Configuration

4.1 Master Node Configuration

Setting up the necessary replication settings in postgresql.conf (e.g., wal_level, max_wal_senders) as shown in the postgresql.conf above.

Configuring “pg_hba.conf” to allow replication connections from slave nodes.

Adding the directives:

```
host    replication     unosm_p_rep     10.128.136.16/32        trust

host    replication     unosm_p_rep     10.128.8.29/32          trust
```

4.2 Slave Node Configuration (Brindisi)

On the slave server in Brindisi “unosm-r-db-02.global.un.org”, run the command:

```
pg_basebackup -h unosm-r-db-01.global.un.org -p 5432 -U unosm_p_rep -c fast -C -S slot_unosm_r_db_02 -D
/var/lib/pgsql/15/data/ -Fp -Xs -P -R -v
```

here the “-h unosm-r-db-01.global.un.org” option specifies the master server from which the backup will be taken, in this case, the server unosm-r-db-01. The “-p 5432” option sets the port to 5432, which is the default PostgreSQL port. The “-U unosm_p_rep” option specifies the user unosm_p_rep for connecting to the PostgreSQL server, which must have the necessary privileges for replication. The “-c fast” option triggers a "fast" checkpoint, ensuring that all data changes are written to disk before the backup starts, minimizing the chance of inconsistencies. The -C option includes the backup label file and restore command in the backup, which is useful for replication setups, allowing a new instance to use the restore command to initiate the restoration process. The “-S slot_unosm_r_db_02” option specifies the replication slot slot_unosm_r_db_02, which is used to retain the WAL files necessary for replication during the backup.

The “-D /var/lib/pgsql/15/data/” option defines the destination directory where the backup will be stored, in this case, /var/lib/pgsql/15/data/, which must be an empty directory or contain a clean data directory. The -Fp option indicates the backup should be taken in plain format, meaning that the files will be copied directly from the PostgreSQL data directory. The -Xs option enables streaming of WAL (Write-Ahead Logs), ensuring that all WAL files are included in the backup for point-in-time recovery (PITR). The -P option enables progress reporting, displaying how much data has been transferred during the backup process. Finally, the -R option creates a recovery.conf file in the backup, which is needed for a replica server to start up and perform replication after the base backup is restored.

On the third slave server “unosm-r-db-03.global.un.org”, as “postgres” user run the command:

```
pg_basebackup -h unosm-r-db-01.global.un.org -p 5432 -U unosm_p_rep -c fast -C -S slot_unosm_r_db_03 -D
/var/lib/pgsql/15/data/ -Fp -Xs -P -R -v
```

in this case you have to change also the slot name using “slot_unosm_r_db_03” instead of “slot_unosm_r_db_02”. All the other parameters are the same.

4.3 Asynchronous Replication Considerations

Asynchronous replication in PostgreSQL is a method where data from the primary (master) server is replicated to one or more standby (replica) servers, but the primary server does not wait for the standby servers to acknowledge receipt of the changes before confirming a transaction to the client. This allows the primary server to continue processing transactions without waiting for replication to complete, improving performance. The primary server generates Write-Ahead Logs (WAL) for every change made to the database. These WAL entries describe the modifications (inserts, updates, deletes) and are stored in a separate area on disk. Once the transaction is committed and the WAL entry is written to disk, the primary server acknowledges the client, indicating that the operation has been completed from its perspective.

Note: due to limited space on the root disk where PostgreSQL is installed, the "pg_wal" directory, which is part of "/var/lib/pgsql/15/data" and contains the Write-Ahead Logs (WAL), has been moved to the "/data" directory. A symbolic link has been created at the original location to redirect to the new directory (see figure above).

The WAL entries are then propagated from the primary server to the standby servers. The primary server continuously streams WAL data to the standby servers via the WAL sender process, while the standby servers receive the WAL logs using the WAL receiver process. The standby servers apply these logs to their local copy of the database, keeping them synchronized with the primary server. However, in asynchronous replication, the primary does not wait for the standby to apply the WAL logs before it acknowledges the transaction to the client. This leads to a delay between when the primary server commits a transaction and when that transaction becomes visible on the standby, due to factors like network latency and the time it takes for the standby to apply the logs.

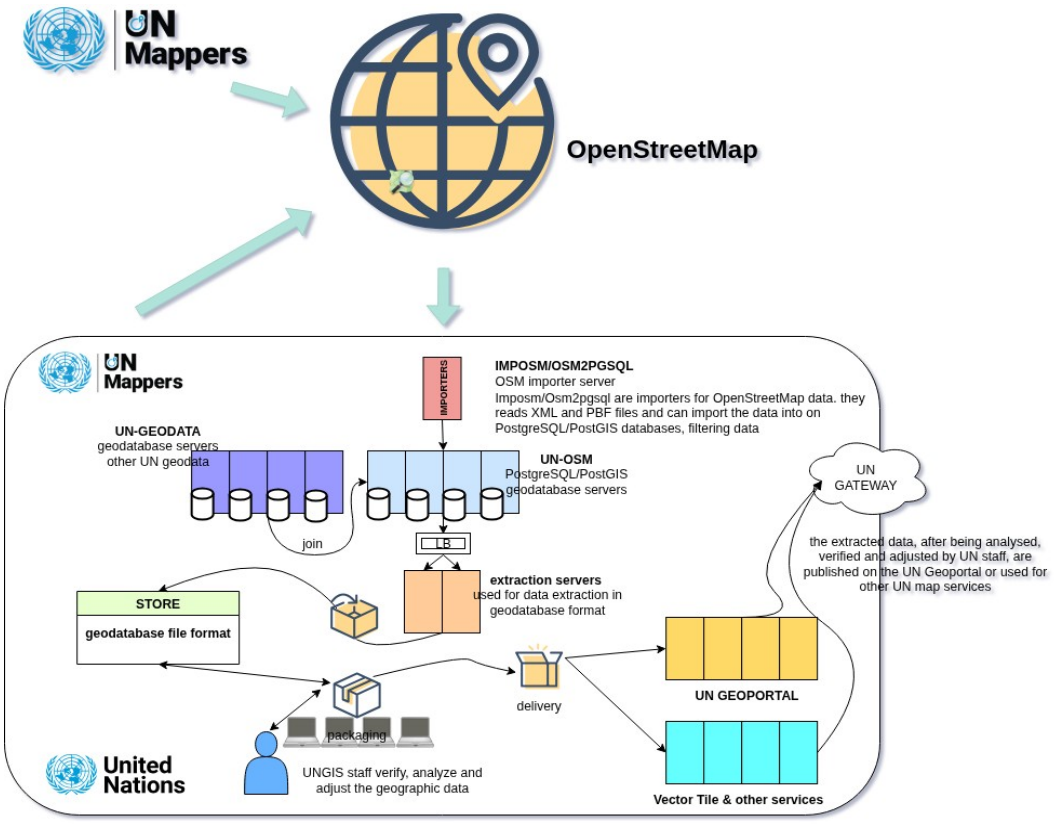
Since asynchronous replication does not require the standby server to confirm the receipt of WAL data before the primary acknowledges the commit, there is a risk of data loss in the event of a primary server failure. If the primary crashes before the standby has received or applied the latest WAL entries, some recent transactions might not be replicated, leading to potential data inconsistency. Asynchronous replication focuses on performance, allowing the primary server to handle transactions quickly, while the standbys catch up with the changes at their own pace.

5. Architecture

UN-OSM is a database platform built on PostgreSQL 15 and hosted on the UNGSC Private Cloud. This architecture comprises a master node and three slave nodes located in the Brindisi data center. The databases on this platform leverage OpenStreetMap (OSM) data, which is downloaded in "pbf" format from the OSM website. The data is imported into the database using the Imposm importer, a tool designed for efficient handling of large OSM datasets (more information about Imposm can be found in its official documentation here https://imposm.org/docs/imposm3/latest).

5.1 Logical Architecture

The following diagram illustrates the logical architecture of the UN-OSM platform, showcasing the relationships between the master and slave nodes, as well as the data flow from OpenStreetMap to the



As illustrated in the image above, geographic data is downloaded from OpenStreetMap (OSM) in "pbf" format and subsequently loaded into the UN-OSM platform using the Imposm importer. Imposm operates on a dedicated server, which is detailed in a summary table within the documentation. The data from the pbf file is processed by Imposm

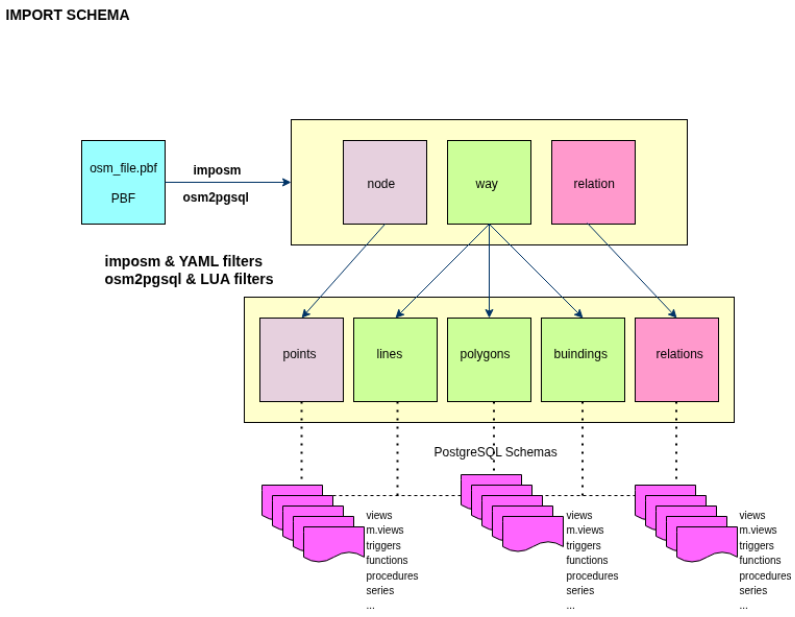
according to the directives specified in a YAML configuration file. This processing populates the tables within the osm_planet schema. From these tables, a comprehensive set of views, table, functions, triggers, store procedures and materialized views is created, enabling data extraction based on a processing scheme that has evolved over time.

The data is accessible through direct connections to the database using tools such as ArcGIS Pro or QGIS.

Additionally, extraction scripts utilizing GDAL libraries are defined on the extraction servers, facilitating further data handling. Once the data is extracted, it undergoes analysis and cleaning by the GIS Analyst data team. After this process, the cleaned data is reassigned to the GIS Visualization team for various applications as outlined below.

5.2 Database Schema

The current OpenStreetMap database schema primarily consists of five main tables: points, lines, polygons, buildings, and relations.



From these core tables, various views are developed to extract and present data in different user-defined schemas for a wide range of applications. This includes services that utilize Vector Tiles for near real-time updates, ESRI Vector Tiles for world maps, on-demand mappers via the MoD service, and support for UN missions like OCHA and MONUSCO. For a detailed view of the database structure, one can directly access the database and examine its schema

6. High Availability, Failover and Disaster Recovery

Regarding the Disaster Recovery service in Valencia, it is currently unavailable due to insufficient physical resources; virtual machines have been requested, but have not yet been assigned due to resource limitations. To enable production capabilities at the Valencia site, it will be necessary to set up a service mirroring the one in Brindisi, possibly on a smaller scale, capable of managing Disaster Recovery operations in the event of problems at the main site.

7. Backup and Recovery Strategy

Since data can be imported from the OpenStreetMap hosting platform in a matter of days, daily backups are not necessary. Instead, the database can be rebuilt directly from OSM. This process should be managed by the application team responsible for maintaining the geography service.

This approach is consistent with best practices for managing large OpenStreetMap imports, which often involve bulk uploads and periodic updates rather than continuous incremental backups, optimizing both time and resource usage.

8. Troubleshooting and Common Issues

Troubleshooting and common issues in PostgreSQL can arise from various factors, such as configuration errors, resource limitations, or replication problems. One common issue is replication lag, where the standby node falls behind in applying Write-Ahead Logs (WAL), potentially leading to data inconsistencies. This can be addressed by optimizing replication settings or increasing resources on the standby server. Another common problem is connection issues, often caused by incorrect settings in the pg_hba.conf file or network problems.

Performance degradation can also occur due to inefficient queries, lack of proper indexing, or insufficient disk space. Monitoring tools and logs, such as pg_stat_activity and PostgreSQL's log files, can be helpful in identifying the root causes of issues. Addressing these problems typically involves adjusting configuration parameters, ensuring proper hardware resources, and optimizing queries to maintain the stability and performance of the database system.

9. Monitoring, Management and Maintenance

Effective monitoring, management, and maintenance are essential components for ensuring the reliability and performance of a PostgreSQL 15 database cluster. In a setup consisting of three nodes—one master and two slaves, with one located on-site and the other in a remote location for disaster recovery—these practices become even more critical. This section will explore the strategies and tools necessary for monitoring system health, managing resources, and performing routine maintenance tasks. By leveraging built-in PostgreSQL features alongside external monitoring solutions, administrators can proactively identify performance bottlenecks, ensure data integrity, and facilitate seamless failover operations in case of node failures.

9.1 Monitoring

Monitoring the status of a PostgreSQL 15 database is essential for maintaining optimal performance and ensuring the reliability of your data environment. To effectively monitor PostgreSQL, administrators can utilize various metrics and tools that provide insights into system health and performance. Key areas to focus on include query performance, where tools like pg_stat_statements can help identify slow or frequently executed queries that may need optimization. Additionally, monitoring memory usage—such as shared_buffers and work_mem—is crucial for understanding how well the database is utilizing resources. It's also important to keep an eye on disk I/O metrics, including read and write operations per second, as excessive I/O can significantly impact performance. Regularly checking connection metrics—active versus idle connections—can help prevent bottlenecks caused by too many simultaneous connections. Furthermore, leveraging logging features allows for detailed analysis of database activities, helping to identify issues like long-running transactions or failed queries. By implementing these monitoring practices, database administrators can proactively manage their PostgreSQL environment, ensuring it operates smoothly across all nodes in the cluster.

Monitor database status: check if database is operational

systemctl status postgresql-15

Monitor replication: check replication status on the new master to ensure the old master is syncing correctly as a standby.

on master node: *select * from pg_stat_replication;*
on slave node: *select * from pg_stat_wal_receiver;*

Monitor database activity: check database activity

SELECT query, calls, total_exec_time, rows,100.0 * shared_blks_hit / NULLIF(shared_blks_hit + shared_blks_read, 0) AS hit_percent FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 10;

Monitor the FS and database logs: check OS filesystems and database logs

The databases are organized within a tablespace located in the "/data" filesystem, specifically under the "/data/database" directory. This designated folder is intended to host the various databases utilized by the multi-purpose UN-OSM platform and requires vigilant monitoring to prevent data corruption due to overfilling or other issues. To ensure the integrity and performance of the databases, it is crucial to keep an eye on database logs and several system metrics, including disk I/O activity, SWAP usage, and other operating system parameters. Monitoring these aspects helps identify potential bottlenecks that could affect database performance. Classic Linux tools such as sar, vmstat, and top can be employed for this purpose. These utilities provide valuable insights into system resource utilization, allowing administrators to take proactive measures to maintain optimal database operation. Regular monitoring not only safeguards against data loss but also enhances overall system efficiency, ensuring that the UN-OSM platform operates smoothly and effectively.

9.2 Database Management

The commands below provide a comprehensive way to manage the PostgreSQL service, facilitating efficient database administration.

Start PostgreSQL Service: To start the PostgreSQL 15 service, use the command:

sudo systemctl start postgresql-15

Stop PostgreSQL Service: To stop the PostgreSQL 15 service, execute:

sudo systemctl stop postgresql-15

Restart PostgreSQL Service: To restart the PostgreSQL service, which is useful after making configuration changes, run:

sudo systemctl restart postgresql-15

Reload PostgreSQL Configuration: To reload the PostgreSQL service configuration without stopping it, use:

sudo systemctl reload postgresql-15

Check Status of PostgreSQL Service: To monitor the current status of the PostgreSQL service, use:

sudo systemctl status postgresql-15

Enable PostgreSQL to Start at Boot: To ensure that PostgreSQL starts automatically when the system boots, execute:

sudo systemctl enable postgresql-15

Disable Automatic Start on Boot: If you need to prevent PostgreSQL from starting at boot, run:

sudo systemctl disable postgresql-15

9.3 Manage monthly patching service

Before applying the monthly patches to the database, ensure that the data import services are not actively writing to the database. To verify this, run the following commands on the “unosm-r-app-01”:

systemctl status imposm_upd_osm_planet.service

systemctl status imposm_upd_osm_planet_buildings.service

If the log is frozen for a few minutes, it means the services have stopped writing to the database. You'll then see the number of diffs applied and the time it took to apply them. In this case, you're ready to apply the OS patches and restart the server/database. As usual, we recommend to stop the database on the slave nodes and then on the master node, then applying the OS patches on the nodes, and finally restarting the master node and then (once the master database is up and running) the slave nodes.

10. Approval

This section acknowledges the acceptance of this document by the Project Board.

	Name	Signature	Date (DD/MM/YYYY)
Project Executive	Type here.		Select a date.
Senior User	Type here.		Select a date.
Senior Supplier	Type here.		Select a date.