Systems and Database Administration

C20437052

TU Dublin

TU-857/4



**Scenario: Stock Broker**

# Security

## Discussion

In this section we are going to discuss the security measures we have taken for this database system based around Stockbrokers. The main things that will be discussed are the major risks and challenges that could happen to this system, the measures that are going to be taken to combat these risks and then the actual policies that should be in place after deployment to combat these risks also. Then I will discuss and show you how these security measures can be implemented step by step.

### Major Risks and Challenges

One of the main risks and challenges with any database system is a user that shouldn’t have access at all gets into the system and access something they shouldn’t be able to access. This could also be caused by some form of malicious threat or by complete accident of someone internally getting access to something they are not intended to get access to. Data breaches are also a potential risk in this system which would cause a problem as the loss of information would cause big financial loses and also damage the reputation of this stockbroker company. The next big possible risk is traders or employees possibly could compromise certain security methods depending on the access they have to the system.

### Measures taken to combat risks and challenges

To combat the possibility of the users having the wrong access we will implement role based access control (RBAC) which would make the system create a role for the trader and the customer and then provide only the appropriate permissions to those roles that they would need to make the system function as intended. Some of these measures would be giving the traders access to select, insert and update on all certain tables and providing only read only access to the role of the customer on certain tables. Sensitive data would be encrypted when at rest and in transit to limit the damage that a data breach would cause to the system. Scheduled security audits are conducted to identify vulnerabilities and apply patches and updates promptly.

### Recommended post-deployment policies

For the policies that should be implemented after the deployment of the system the easiest recommendation for this would be that all employees, including traders and database administrators would need to undergo regular security training to stay updated on the best practices and the possible threats that could occur throughout the system. Next, we would implement a regular review of user access rights and permissions. And these should continue to maintain strict access control policies across the system. A detailed incident response plan should be in place to address security breaches promptly and efficiently. The whole idea for these would be design an if this happens then do this for the administrators of the database to follow in case of any issues occurring.

## Step-by-step

### Enforce Role-Based-Access Control (RBAC)

-- Create roles for Traders and Customers

CREATE ROLE Traders LOGIN;

CREATE ROLE Customers LOGIN;

-- Grant appropriate permissions to Traders

GRANT SELECT, INSERT, UPDATE ON Companies TO Traders;

GRANT SELECT, INSERT, UPDATE ON Prices TO Traders;

GRANT SELECT, INSERT, UPDATE ON Positions TO Traders;

-- Grant read-only access to Customers

GRANT SELECT ON Positions TO Customers;

### Enabling SSL and SSL Encryption:

By default, SSL support may be disabled in PostgreSQL. You need to ensure that SSL support is enabled in the PostgreSQL build or package you are using. Most modern distributions and installations include SSL support by default, but you may need to verify this.

Obtain SSL/TLS certificates from a trusted Certificate Authority (CA) or generate self-signed certificates if you don't have a CA. Generate a server certificate (server.crt) and a server private key (server.key). Optionally, obtain a Certificate Authority (CA) certificate (ca.crt) if you want to verify the server's identity using a trusted CA. To generate the certificates needed for this system run this code in the data directory of the system.

openssl req -nodes -new -x509 -keyout server.key -out server.crt -subj

‘/C=IE/L=Dublin/O=TUDublin/CN=postgres’

Locate the PostgreSQL configuration file (postgresql.conf). This file is usually found in the PostgreSQL data directory. Open the postgresql.conf file in a text editor. Uncomment or add the following lines to enable SSL and specify the paths to your certificate files: It should look like this.

A computer screen shot of a program

Description automatically generated

Open the pg\_hba.conf file, which is also located in the PostgreSQL data directory. For each client connection type (local, host, hostssl, hostnossl), specify the authentication method (trust, md5, password, etc.) and indicate whether SSL is required or not. For SSL-enabled connections, use the 'hostssl' line type and specify 'require' as the authentication method to enforce SSL/TLS encryption. This should look like this only allowing ssl from this specified ip address connection

A screenshot of a computer

Description automatically generated

Then all that is needed is to restart postgress and then test to see if this worked by attempting to connect to the database through SSL. To restart the database run:

sudo systemctl restart postgresql

### Scan the database system for any vulnerabilities

Metasploit is a widely-used penetration testing framework that includes a variety of tools and modules for identifying and exploiting security vulnerabilities. You can use Metasploit to perform penetration testing against PostgreSQL servers to identify weaknesses and assess the overall security posture. Example command to run Metasploit against a PostgreSQL server:

msfconsole

use auxiliary/scanner/postgres/postgres\_login

set RHOSTS postgresql\_server\_ip

exploit

Below is a picture of that command being run

A screenshot of a computer

Description automatically generated

# Auditing

## Discussion

### Reasons to audit the database

Auditing is very important in the management of a database as it can ensure security, integrity and compliance of a database system. It involves monitoring and recording database activities so there is a log of exactly what happened on the system when. The main objectives of auditing are security, compliance , forensic analysis and performance monitoring. The security aspect is broken down as it can track and display any possible login attempts and unauthorized actions on the system. For compliance auditing will ensure that it follows the standards of the industry in its current form as there is an audit trail of the data access and modifications to the system. For forensic analysis it would capture the details of any actions taken in case of a security breach or data breach the breached information would then be known to the database administrators by capturing the details of user activities and the system events. The performance monitoring can be done as the auditing could also track resource usage and query execution times of the database.

### Audit configuration for this database.

To support the objectives of the auditing policy, the database will be configured to enable auditing features and define audit policies. PostgreSQL provides several options for auditing, including:

1. PostgreSQL supports standard logging of database activities to log files. This includes logging of SQL statements, errors, and connection events.
2. pgAudit is a popular extension for PostgreSQL that provides more advanced auditing capabilities, including detailed logging of user actions, data modifications, and object access.
3. PostgreSQL allows you to create custom audit triggers on tables to capture specific data changes, such as INSERT, UPDATE, and DELETE operations.
4. PostgreSQL also supports database-level auditing using event triggers, which allow you to capture database-wide events such as user login/logout and DDL statements.

For this scenario, we will choose to implement auditing using the pgAudit extension. pgAudit offers comprehensive auditing capabilities, including detailed logging of user actions and data modifications, which aligns well with the objectives of the auditing policy. Additionally, pgAudit provides fine-grained control over audit policies, allowing us to customize auditing rules based on specific requirements.

## Step-by-step

Use the appropriate package manager or download the pgAudit extension from the PostgreSQL extension repository.

Install the pgAudit extension using the following command in SQL:

CREATE EXTENSION IF NOT EXISTS pg\_audit;

Configure pgAudit settings in the PostgreSQL configuration file (postgresql.conf) to specify the audit log destination, format, and filtering options. Open the postgresql.conf file in a text editor. Add or modify the following settings

pgaudit.log = 'all';

pgaudit.log\_catalog = on;

pgaudit.log\_level = 'log';

Save the changes and restart the PostgreSQL server to apply the new configuration.

Run these then to set up audit policies for all the tables:

SELECT audit.audit\_table('public.companies', 'all');

SELECT audit.audit\_table('public.traders', 'all');

SELECT audit.audit\_table('public.portfolios', 'all');

Grant the correct privileges on the audit functions to the trader role using this command:

GRANT EXECUTE ON FUNCTION audit.audit\_table TO traders;

Then if you want to look at all the audit logs run the command

SELECT \* FROM pg\_audit\_log;

# Performance Optimisation

## Discussion

### Possible bottlenecks and issues

Query performance is very important as slow running queries is one of the most common performance bottlenecks that occur in databases. The goal would be to identify queries with high execution time, frequent table scans or inefficient execution plans and attempt to limit these issues. Indexing is also a possible bottleneck as inadequate or missing indexes can lead to slow query performance especially for frequently executed select queries. Locking and blocking is where multiple transactions compete for the same resources leading to one taking said resources causing the other transaction to be blocked out and thus causing a decreased throughput.

### Options for performance Optimization

For the query optimization that would involve rewriting the slower running queries to improve the performance. The way to do this can be done by adding indexes creating partitioned tables but also using more efficient join methods or optimizing the where clauses can help improve performance. The index optimization as well as just having indexes in the first place around frequently executed queries would improve the performance of the database. Tune PostgreSQL configuration parameters such as shared\_buffers, work\_mem, and effective\_cache\_size to optimize memory usage and query performance. Adjust parameters based on workload characteristics and available system resources. Next, implementing connection pooling can reduce the overhead of establishing and tearing down database connections. This could improve the applications scalability and performance by reusing existing database connections.

### Chosen Options and Why

1. Table Partitioning:

We will prioritize implementing table partitioning to improve database performance by distributing large tables into smaller, more manageable partitions based on predefined criteria such as date or portfolio ID. This approach addresses scalability issues and enhances query performance by reducing the size of individual partitions and enabling partition pruning. Partitioning tables based on predefined criteria such as date or portfolio ID allows us to distribute data across multiple partitions, reducing the size of individual partitions and improving query performance. This approach enhances database scalability and manageability, particularly for large tables with high volumes of data.

1. Index Optimization:

We will also optimize indexes by reviewing and adjusting existing indexes to align with the partitioning strategy. By creating appropriate indexes on partitioned tables, we can further enhance query performance and access efficiency. Optimizing indexes on partitioned tables ensures that query performance benefits from both partition pruning and efficient index access. By aligning index usage with the partitioning strategy, we can further optimize data retrieval and query execution, leading to improved overall database performance.

## Step-by-step

### Indexing

Create an index on the companyID column, as it is likely used in JOIN operations or WHERE clauses to retrieve company-specific data.

CREATE INDEX idx\_companies\_companyID ON public.companies (companyID);

Depending on the access patterns, you may consider indexing the staff ID column if it's frequently used in queries to retrieve trader information.

CREATE INDEX idx\_traders\_staffID ON public.traders (staffID);

If portfolios are frequently queried based on portfolio ID, create an index on the portfolioID column.

CREATE INDEX idx\_portfolios\_portfolioID ON public.portfolios (portfolioID);

If historical prices are queried based on companyID or date, create indexes on these columns for efficient data retrieval.

CREATE INDEX idx\_prices\_companyID ON public.prices (companyID);

CREATE INDEX idx\_prices\_date ON public.prices (date);

Depending on the queries performed on the Positions table, you may consider indexing the companyID and portfolioID columns

CREATE INDEX idx\_positions\_companyID ON public.positions (companyID);

CREATE INDEX idx\_positions\_portfolioID ON public.positions (portfolioID);

### Partitioning Tables:

Partition the Prices table based on the date column to store historical prices in separate partitions for each time period (e.g., monthly or yearly).Create partitioned tables for each time period, such as prices\_2022, prices\_2023, etc. Below is an example of the code to do so it would be recommended to do this for all the years possible. This could even be split per quarter of a year but by year for now should do fine. Below is also code to partition by quarters of a year in this case 2023

--For yearly based partitions

CREATE TABLE prices\_2022 PARTITION OF public.prices FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');

CREATE TABLE prices\_2023 PARTITION OF public.prices FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

--for Quarter partitions

CREATE TABLE public.prices\_q1 PARTITION OF public.prices

FOR VALUES FROM ('2022-01-01') TO ('2022-04-01');

CREATE TABLE public.prices\_q2 PARTITION OF public.prices

FOR VALUES FROM ('2022-04-01') TO ('2022-07-01');

CREATE TABLE public.prices\_q3 PARTITION OF public.prices

FOR VALUES FROM ('2022-07-01') TO ('2022-10-01');

CREATE TABLE public.prices\_q4 PARTITION OF public.prices

FOR VALUES FROM ('2022-10-01') TO ('2023-01-01');

Partition the Positions table based on the portfolioID column to store positions data for each portfolio in separate partitions.Create partitioned tables for each portfolio, such as positions\_portfolio\_1, positions\_portfolio\_2, etc. Below is example code for this:

CREATE TABLE positions\_portfolio\_1 PARTITION OF public.positions FOR VALUES IN ('portfolio\_1');

CREATE TABLE positions\_portfolio\_2 PARTITION OF public.positions FOR VALUES IN ('portfolio\_2');

-- Create additional partitions for other portfolios as needed

# Backup/Recovery/Availability

## Discussion

### Objectives for backup and recovery

For data protection we must ensure the protection and integrity of critical data stored in the database against accidental loss, corruption, or hardware failures. Next to think of would be disaster recovery which would enable the quick recovery of data in the event of a disaster such as a server failure, data corruption, accidental deletion. To minimize downtime and ensure business continuity by having reliable backup and recovery mechanisms in place to restore the operations as fast as possible.

### Options available for backup and recovery

Taking physical backups by using tools such as pg\_basebackup or file system-level backups. These backups capture the entire database cluster including such things as the data files, configuration files and transaction logs. Using the pg\_dump to create logical backups that contain SQL statements to recreate the database schema and the data. Logical backups offer flexibility but may take longer to restore when compared to the other options available. Continuous Archiving and Point-in-Time Recovery(PITR) would enable recovery to a specific point in time minimizing data loss in case of failure.

### Chosen approach and Why

We have chosen to implement logical backups using pg\_dump for our database. This approach provides a flexible and portable backup solution, allowing us to create SQL-format backups that contain schema and data information. The main benefits of this is the flexibility as this allows for the selective backup and restoration of specific database objects making it work for partial backups or data migration scenarios. It is also very easy to use and set up as pg\_dump is already there built into PostgreSQL and it requires minimal configuration changes. The only real downside to this is that the logical backups of this can be slower when compared to physical backups especially for larger databases due to the SQL statements need to be generated for each and every database object.

## Step-by-step

To create a backup using pg\_dump run the command

pg\_dump -U username -d dbname -f backup.sql

For my example it looks more like this as you must change the username for the username of the database and the dbname for the actual name of the database.

pg\_dump -U c20437052 -d stockbroker -f stockbroker\_backup.sql

To automate the backup process you can run a command with cron job to run pg\_dump daily at midnight

0 0 \* \* \* pg\_dump -U c20437052 -d stockbroker -f /path/to/backup/stockbroker\_backup\_$(date +\%Y\%m\%d).sql

Then to restore a backup you would run this command

psql -U c20437052 -d new\_stockbroker\_database -f stockbroker\_backup.sql

This command restores the backup to a new database named "new\_stockbroker\_database" (replace with your desired database name).