Systems and Database Administration

<YOUR STUDENT NUMBER>

TU Dublin

TU-857/4



# Security

### Major Risks and Challenges

The stockbroker database faces several significant security risks:

1. **Unauthorized Access**: The database contains sensitive financial information including portfolio positions and trader details, making it a high-value target for attackers.
2. **Data Theft and Insider Threats**: Traders with legitimate access could potentially extract or manipulate data for personal gain, particularly regarding company positions data.
3. **Network Vulnerabilities**: Customers can access the database remotely, creating potential network attack vectors.
4. **Privilege Escalation**: Users might attempt to gain higher privilege levels than intended, especially concerning the positions table which contains actionable financial data.
5. **SQL Injection**: Both the customer-facing and trader interfaces could be vulnerable to SQL injection if proper safeguards aren't implemented.
6. **Lack of Encryption**: Unencrypted data both in transit and at rest could be intercepted or accessed through unauthorized means.

### Measures taken to combat risks and challenges

To address these security concerns, I've implemented several protective measures:

1. **Role-Based Access Control (RBAC)**: Creating specific roles for traders and customers with precisely defined permissions.
2. **Network Access Restrictions**: Configuring pg\_hba.conf to restrict trader access to local connections only, while allowing customer access from specific IP ranges.
3. **SSL/TLS Encryption**: Implementing encrypted connections to protect data in transit, particularly for customer access.
4. **Password Policies**: Enforcing strong password requirements and password rotation for database users.
5. **Row-Level Security (RLS)**: Implementing RLS policies to ensure customers can only view positions related to their portfolios.
6. **Database Encryption**: Enabling encryption for sensitive data at rest.

### Recommended post-deployment policies

After deployment, the following security policies should be maintained:

1. **Regular Security Audits**: Conduct quarterly security assessments to identify potential vulnerabilities.
2. **User Access Reviews**: Perform monthly reviews of user access rights to ensure adherence to the principle of least privilege.
3. **Patch Management**: Maintain a regular schedule for applying security patches to the PostgreSQL database.
4. **Security Training**: Provide ongoing training for database administrators and users regarding security best practices.
5. **Incident Response Plan**: Develop and maintain a database security incident response procedure.
6. **Security Monitoring**: Implement real-time monitoring for suspicious activities or unauthorized access attempts.

### Step-by-step

**First, create a new database directory for the stockbroker database:**

* sudo mkdir -p /home/postgres/stockbroker\_db
* sudo chown postgres:postgres /home/postgres/stockbroker\_db
* sudo -u postgres initdb -D /home/postgres/stockbroker\_db

### Enabling SSL

A screen shot of a computer screen

AI-generated content may be incorrect.

Figure 1- Generate SSL certificates

**Edit postgresql.conf to enable SSL:**

* sudo -u postgres nano /home/postgres/stockbroker\_db/postgresql.conf
* ssl = on
* ssl\_cert\_file = 'server.crt'
* ssl\_key\_file = 'server.key'

**Edit pg\_hba.conf to restrict access based on roles:**

* sudo -u postgres nano /home/postgres/stockbroker\_db/pg\_hba.conf

**Start PostgresSQL server, create the database and setup roles and then connect to the databsebase**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**Commands**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

# Auditing

## Roles of Auditing in the stockbroker database

Auditing plays a critical role in the stockbroker database environment for several key reasons:

1. **Regulatory Compliance**: Financial institutions, including stockbrokers, are subject to strict regulatory requirements such as SEC regulations, SOX compliance, and financial industry standards that mandate comprehensive audit trails of all financial transactions.
2. **Fraud Detection**: An effective audit system can identify unusual patterns or suspicious activities that might indicate fraud, market manipulation, or insider trading.
3. **Data Integrity Verification**: Auditing ensures the accuracy and integrity of financial data, which is crucial in the high-stakes environment of stock trading where even minor errors can have significant financial implications.
4. **Accountability**: Audit logs create a chain of accountability by tracking which trader performed which action, when it was performed, and what data was modified.
5. **Forensic Analysis**: In case of security incidents or disputes, audit logs provide essential forensic evidence to investigate what happened, how it happened, and who was responsible.

## Objectives of the Auditing Policy

For our stockbroker database, the auditing policy aims to achieve the following objectives:

1. **Complete Transaction Tracking**: Record all modifications to the Positions table, which represents the actual financial holdings and is therefore the most critical data to track.
2. **User Activity Monitoring**: Track all login attempts (both successful and failed) to identify potential unauthorized access attempts.
3. **Privileged Operations Logging**: Monitor administrative actions like role changes, permission modifications, or schema alterations that could affect security.
4. **Query Logging for Sensitive Data**: Record access to sensitive financial data, particularly for the Positions and Portfolios tables.
5. **Immutable and Tamper-Evident Logs**: Ensure audit logs cannot be modified or deleted by regular users, maintaining their integrity as legal evidence if required.
6. **Performance Balance**: Implement auditing with minimal impact on database performance, particularly during high-volume trading periods.

## Available Auditing Options in PostgreSQL

PostgreSQL offers several approaches to auditing:

1. **PostgreSQL's Built-in Logging**:
   * Basic logging capabilities through postgresql.conf settings
   * Can log statements, connections, and disconnections
   * Limited in granularity and customization
2. **Trigger-Based Auditing**:
   * Custom triggers on tables to track changes
   * Highly customizable but can impact performance
   * Stores audit information in separate tables
3. **pgAudit Extension**:
   * Dedicated extension for comprehensive auditing
   * Provides session and object audit logging
   * Configuration through postgresql.conf and specific role settings
   * More efficient than trigger-based approaches for high-volume auditing
4. **External Solutions**:
   * Third-party tools like Audit Trigger by 2ndQuadrant
   * Log management systems that can parse PostgreSQL logs

## Chosen Auditing Solution

For our stockbroker database, I've chosen a hybrid approach:

1. **pgAudit Extension** as the primary auditing mechanism:
   * Provides comprehensive, high-performance auditing capabilities
   * Minimal impact on database operations compared to trigger-based solutions
   * Configurable logging levels to balance detail with performance
2. **Supplemental Custom Triggers** for the Positions table:
   * Captures detailed before/after values for all trade-related changes
   * Stores this information in a separate audit table with timestamp and user information
   * Preserves specific business context that pgAudit might not capture

This hybrid approach provides the best balance of:

* Performance efficiency during high-volume trading
* Comprehensive coverage of all security-relevant events
* Detailed tracking of financial position changes
* Compliance with regulatory requirements

## Step-by-step

**Install pgAudit on Ubuntu**

* sudo apt-get update
* sudo apt-get install postgresql-13-pgaudit

**Edit the PostgreSQL configuration file to enable and configure pgAudit:**

* sudo -u postgres nano /home/postgres/stockbroker\_db/postgresql.conf

A screenshot of a computer program

AI-generated content may be incorrect.

**Create the Log Directory**

* sudo -u postgres mkdir -p /home/postgres/stockbroker\_db/pg\_log
* sudo -u postgres chmod 700 /home/postgres/stockbroker\_db/pg\_log

# Performance Optimisation

## Most likely Performance Bottleneck and Potential Issues

The stockbroker database will face several performance challenges due to its nature and usage patterns:

1. **End-of-Day Position Updates**: The scenario states that at the end of each trading day, all traders submit their trades which need to be processed as quickly as possible. This creates a concentrated burst of write operations to the Positions table.
2. **Historical Price Data Growth**: The Prices table will grow continuously as new price data is added daily for each company. With 2000+ companies and daily price entries, this table will rapidly accumulate millions of records.
3. **Analytical Queries on Historical Data**: The scenario mentions that the Prices table "will occasionally be used to generate reports and provide large batches of data for machine learning algorithms," indicating heavy analytical queries that could strain the system.
4. **Concurrent Customer Queries**: Since customers have read-only access to view their portfolio positions and may integrate with their own applications, there could be numerous concurrent read requests.
5. **Join-Heavy Queries**: Meaningful queries will likely involve joins across multiple tables (e.g., Positions joined with Companies and Portfolios), which can be resource-intensive without proper optimization.
6. **Index Maintenance Overhead**: While indexes improve read performance, they add overhead to write operations, which could impact the critical end-of-day position updates.

## Available Performance Optimization Options

PostgreSQL offers several performance optimization techniques relevant to our scenario:

1. **Indexing Strategies**:
   * B-tree indexes for equality and range queries
   * Hash indexes for simple equality operations
   * Partial indexes to focus on relevant subsets of data
   * Multicolumn indexes for frequently combined query conditions
   * Covering indexes (including columns in INCLUDE) to avoid table lookups
2. **Partitioning**:
   * Range partitioning for time-series data (Prices table)
   * List partitioning for categorical data
   * Hash partitioning for evenly distributed load
3. **Memory Configuration**:
   * Optimizing shared\_buffers for caching frequently accessed data
   * Effective work\_mem sizing for complex sort and join operations
   * Appropriate maintenance\_work\_mem for maintenance operations
4. **Query Optimization**:
   * Creating materialized views for common analytical queries
   * Using VACUUM and ANALYZE to maintain statistics and reclaim space
   * Implementing connection pooling to handle concurrent connections efficiently
5. **Hardware Optimizations**:
   * Utilizing SSDs for improved I/O performance
   * Separating tablespaces across different storage devices

## Chosen Implementation and Justification

For the stockbroker database, I've chosen to implement the following performance optimizations:

1. **Strategic Indexing**:
   * Create targeted indexes to support the most common query patterns
   * Use covering indexes for frequently accessed data
2. **Table Partitioning for Historical Prices**:
   * Implement range partitioning by date for the Prices table
   * This will significantly improve query performance for time-based analysis
3. **Memory Configuration Tuning**:
   * Optimize PostgreSQL memory parameters based on the server's available resources
   * Focus on supporting the end-of-day trade processing workload
4. **Materialized Views for Analytics**:
   * Create materialized views for common analytical queries on historical data
   * Schedule regular refreshes during off-peak hours
5. **VACUUM and ANALYZE Automation**:
   * Configure autovacuum parameters specifically for the high-write tables
   * Schedule manual VACUUM ANALYZE operations during non-trading hours

## **Benefits and Drawbacks of Chosen Methods**

1. **Table Partitioning for Prices**:
   * **Benefits**: Dramatically improves query performance for date-range queries; enables efficient archiving of older data; allows parallelized operations on different partitions.
   * **Drawbacks**: Increases complexity in database management; may require partition pruning awareness in application queries.
2. **Strategic Indexing**:
   * **Benefits**: Significantly accelerates read operations; improves JOIN performance; reduces full table scans.
   * **Drawbacks**: Increases storage requirements; adds overhead to write operations which could impact end-of-day processing.
3. **Memory Configuration Tuning**:
   * **Benefits**: Better utilization of available server resources; improved caching of frequently accessed data.
   * **Drawbacks**: Requires careful balancing to avoid memory pressure on the server; needs periodic review as data volume grows.
4. **Materialized Views**:
   * **Benefits**: Dramatically improves performance for complex analytical queries; reduces load during business hours.
   * **Drawbacks**: Data is not real-time and has refresh overhead; increases storage requirements.
5. **Vacuum and Analyze Automation**:
   * **Benefits**: Maintains optimal performance over time; prevents bloat in high-churn tables.
   * **Drawbacks**: Regular maintenance operations consume resources; requires careful scheduling to avoid impact on critical operations.

## Step-by-step

Implementing Table Partitioning for Prices Table

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen with white text

AI-generated content may be incorrect.

Creating Partition Maintenance Functions

# Backup/Recovery/Availability

## What are the objectives / recovery policy?

For the stockbroker database, the backup and recovery policy must address several critical requirements:

1. **Data Integrity Protection**: Financial transaction data must be preserved with absolute integrity, as any corruption or loss could result in significant financial implications.
2. **Regulatory Compliance**: Financial institutions are subject to strict regulatory requirements that mandate data retention, recoverability, and auditability of historical transactions.
3. **Minimal Downtime**: Trading activities cannot tolerate extended outages. The database must be recoverable quickly to minimize financial losses.
4. **Point-in-Time Recovery**: The ability to restore the database to a specific moment is essential for financial reconciliation, especially in cases of erroneous trades.
5. **Disaster Recovery Readiness**: Protection against catastrophic events affecting primary data centers, requiring geographic redundancy.
6. **End-of-Day Protection**: Special protection for the critical end-of-day position update process to ensure no data loss during this crucial operation.

## What options are available for backup/recovery?

PostgreSQL offers several approaches to backup and recovery:

1. **Physical Backup Methods**:

* **pg\_basebackup**: Creates a binary copy of the database cluster files
* **Continuous Archiving (WAL Archiving)**: Combines a base backup with Write-Ahead Log archiving for point-in-time recovery

1. **Logical Backup Methods**:

* **pg\_dump/pg\_dumpall**: Creates SQL script files to reconstruct the database
* **Table-specific dumps**: Targeted backup of critical tables like Positions

1. **Replication Solutions**:

* **Streaming Replication**: Maintains standby servers that continuously apply WAL records
* **Logical Replication**: Replicates specific tables using a publish-subscribe model

1. **Specialized Backup Tools**:

* Barman: Comprehensive backup management solution specifically for PostgreSQL
* pgBackRest: Feature-rich backup and restore solution with compression and parallel operations

## What have you chosen and why

For the stockbroker database, I've designed a multi-tiered approach balancing robust protection with operational efficiency:

1. **Primary Strategy: Continuous Archiving with WAL**

* Configure WAL archiving to retain transaction logs
* Implement daily base backups using pg\_basebackup
* Store archived WAL files on separate storage to enable point-in-time recovery
* This approach allows recovery to any point in time, essential for financial data integrity

1. **Secondary Strategy: Targeted Logical Backups**

* Daily logical backups (pg\_dump) of the Positions table after market close
* Weekly full logical database dumps for added redundancy
* This provides an additional layer of protection for the most critical data

1. **High Availability Implementation: Streaming Replication**

* Maintain a hot standby server using PostgreSQL's streaming replication
* Configure with synchronous\_commit for critical transactions
* This ensures near-zero data loss in server failure scenarios

## Benefits and Drawbacks

1. **Continuous Archiving with WAL**:

* **Benefits**: Enables point-in-time recovery; minimal performance impact during trading hours; supports incremental backups
* **Drawbacks**: More complex to manage; requires additional storage for WAL files; more involved restoration process

1. **Targeted Logical Backups**:

* **Benefits**: Provides object-level recovery options; human-readable format for verification; can selectively restore just the Positions table
* **Drawbacks**: Creates additional server load during backup; slower restoration compared to physical methods

1. **Streaming Replication**:

* **Benefits**: Near real-time redundancy; allows for quick failover; can offload read queries to standby servers
* **Drawbacks**: Requires additional infrastructure; synchronous replication impacts write performance; replicates logical corruption

1. **Measures to Increase Availability**

Beyond backup and recovery, these measures should be implemented to maximize availability:

1. **Hardware Redundancy**:

* Deploy servers with redundant components (power supplies, RAID storage, network interfaces)
* Implement server monitoring with pre-failure alerts

1. **Connection Pooling and Load Balancing**:

* Implement pgBouncer for connection pooling to handle high volumes of customer connections
* Configure load balancing to distribute read queries across standby servers

1. **Maintenance Windows and Rolling Updates**:

* Schedule maintenance during non-trading hours (weekends or overnight)
* Implement rolling updates using replication to minimize downtime

1. **Automated Failover**:

* Deploy Patroni or similar tools to automate failover to standby servers
* Configure with appropriate fencing mechanisms to prevent split-brain scenarios

1. **Comprehensive Monitoring**:

* Implement Prometheus and Grafana for real-time database metrics
* Configure alerts for replication lag, disk space, connection count, and query performance

1. **Geographic Distribution**:

* Maintain at least one standby server in a different data center
* Implement regular testing of disaster recovery procedures