Multi-User Database Setup Guide

Enabling VERA Application for ~10 Concurrent Users

Overview

This guide explains how to enable the VERA application to support approximately 10 concurrent users with a shared database. Currently, the application uses SQLite, which is designed for single-user or light multi-user scenarios. For reliable multi-user support, a centralized database server is required.

Solution Options

Option 1: SQLite on Network Share (Not Recommended)

Description: Place the SQLite database file on a network share accessible to all users.

Pros:

- Minimal code changes required
- No additional server software needed
- Simple setup

Cons:

- SQLite has significant limitations with concurrent writes over network
- Performance degradation with multiple users
- High risk of database corruption
- File locking issues across network
- **Maximum recommended: 3-5 users**
- Not suitable for production with 10 users

Setup:

1. Create network share: `\\server\shared\AppData\`
2. Copy `ap_support.db` to network location
3. Update database path in code
4. Ensure all users have read/write permissions

Verdict: ❌ Not recommended for 10 users

Option 2: PostgreSQL Database Server (Recommended) ☑

Description: Deploy a PostgreSQL database server on a central machine or server. All users connect to this database server.

Pros:

- Designed specifically for concurrent multi-user access
- Excellent performance and reliability
- Proper transaction handling and locking
- Built-in user authentication and permissions
- Scales easily beyond 10 users
- Industry-standard solution
- Free and open-source

Cons:

- Requires database server installation and maintenance
- Moderate code changes needed in application
- Need to learn basic PostgreSQL administration

Recommended For: ☑ 10 users - this is the best solution

Option 3: MySQL Database Server (Alternative)

Description: Similar to PostgreSQL but using MySQL/MariaDB.

Pros/Cons: Very similar to PostgreSQL

Note: PostgreSQL is recommended over MySQL for this use case due to better standards compliance and feature set.

Option 4: REST API Server (Future-Proof)

Description: Create a REST API server that hosts the database and business logic. Desktop application becomes a client.

Pros:

- Most scalable solution
- Can add web interface later
- Centralized business logic
- Better security and access control
- Enables mobile apps in future

Cons:

- Most complex to implement
- Requires significant code refactoring
- Need to host and maintain web server

Recommended For: Future consideration if organization grows or web access needed

---

Recommended Implementation: PostgreSQL Setup

Phase 1: Server Setup (IT Administrator)

Step 1: Install PostgreSQL Server

On Windows Server or dedicated machine:

1. **Download PostgreSQL:**
   - Visit: https://www.postgresql.org/download/windows/
   - Download latest stable version (15.x or 16.x)
   - Run installer
2. **Installation Options:**
   - **Components:** PostgreSQL Server, pgAdmin 4, Command Line Tools
   - **Port:** 5432 (default)
   - **Superuser password:** Choose strong password (save securely!)
   - **Locale:** Default
3. **Verify Installation:**
   - Open pgAdmin 4
   - Connect to local server
   - Should see "PostgreSQL 16" in server list

Step 2: Create Database and User

Using pgAdmin 4:

1. **Create Database:**
   - Right-click "Databases" → "Create" → "Database"
   - Name: `vera_support_db`
   - Owner: postgres
   - Click "Save"
2. **Create Application User:**
   - Right-click "Login/Group Roles" → "Create" → "Login/Group Role"
   - General tab:
     - Name: `vera_app_user`
   - Definition tab:
     - Password: `[choose secure password]`
   - Privileges tab:
     - ☑ Can login
   - Click "Save"
3. **Grant Permissions:**
   - Open Query Tool for `vera_support_db`
   - Run SQL:

```
GRANT ALL PRIVILEGES ON DATABASE vera_support_db TO vera_app_user;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO vera_app_user;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO vera_app_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL ON TABLES TO
vera_app_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL ON SEQUENCES TO
vera_app_user;
```

Step 3: Configure PostgreSQL for Network Access

1. **Edit postgresql.conf:**
   - Location: `C:\Program Files\PostgreSQL\16\data\postgresql.conf`
   - Find line: `#listen_addresses = 'localhost'`
   - Change to: `listen_addresses = '*'`
   - Save file
2. **Edit pg_hba.conf (Authentication):**
   - Location: `C:\Program Files\PostgreSQL\16\data\pg_hba.conf`
   - Add line at end:

```
host    vera_support_db    vera_app_user    0.0.0.0/0    md5
```

   - Save file
3. **Restart PostgreSQL Service:**
   - Open Services (services.msc)
   - Find "postgresql-x64-16"
   - Right-click → Restart
4. **Configure Firewall:**
   - Open Windows Firewall
   - Create inbound rule for TCP port 5432
   - Allow connections from your internal network

Step 4: Test Connection

From another machine on network:

1. Install pgAdmin 4 or use psql
2. Create new server connection:
   - Host: `[server-ip-address]`
   - Port: 5432
   - Database: `vera_support_db`
   - Username: `vera_app_user`

- Password: `[password]`

3. Test connection - should succeed

Phase 2: Database Migration

Step 1: Export Current SQLite Data

Create migration script (`migrate_to_postgresql.py`):

```
import sqlite3
import psycopg2
from psycopg2.extras import execute_values

# SQLite source
sqlite_conn = sqlite3.connect('ap_support.db')
sqlite_conn.row_factory = sqlite3.Row
sqlite_cur = sqlite_conn.cursor()

# PostgreSQL destination
pg_conn = psycopg2.connect(
    host='your-server-ip',
    database='vera_support_db',
    user='vera_app_user',
    password='your-password'
)
pg_cur = pg_conn.cursor()

# Get list of tables
sqlite_cur.execute("SELECT name FROM sqlite_master WHERE type='table';")
tables = [row[0] for row in sqlite_cur.fetchall()]

for table in tables:
    print(f"Migrating table: {table}")

    # Get table schema
    sqlite_cur.execute(f"PRAGMA table_info({table})")
    columns = [row[1] for row in sqlite_cur.fetchall()]

    # Get data
    sqlite_cur.execute(f"SELECT * FROM {table}")
    rows = sqlite_cur.fetchall()

    if rows:
        # Create table in PostgreSQL (you'll need proper CREATE TABLE statements)
        # Insert data
        cols = ','.join(columns)
        placeholders = ','.join(['%s'] * len(columns))
        query = f"INSERT INTO {table} ({cols}) VALUES ({placeholders})"
        execute_values(pg_cur, query, rows)
```

```
pg_conn.commit()
print("Migration complete!")
```

Note: You'll need to create the PostgreSQL table schemas first, matching your SQLite schema.

Step 2: Create PostgreSQL Schema

Run your existing schema creation SQL with PostgreSQL-specific adjustments:

- Change `INTEGER PRIMARY KEY AUTOINCREMENT` to `SERIAL PRIMARY KEY`
- Change `TEXT` to `VARCHAR` or keep as `TEXT`
- Adjust any SQLite-specific syntax

Phase 3: Application Code Updates

Step 1: Install Python PostgreSQL Driver

On all user workstations:

```
pip install psycopg2-binary
```

Step 2: Create Database Configuration File

Create `config/db_config.ini`:

```
[database]
type = postgresql
host = 192.168.1.100
port = 5432
database = vera_support_db
user = vera_app_user
password = your-secure-password

[database_dev]
type = sqlite
file = ap_support.db
```

Step 3: Update `database_manager.py`

Add PostgreSQL support to DatabaseManager class:

```
import configparser
import psycopg2
from psycopg2.extras import RealDictCursor
```

```
class DatabaseManager:
    def __init__(self, config_file='config/db_config.ini'):
        config = configparser.ConfigParser()
        config.read(config_file)

        self.db_type = config.get('database', 'type', fallback='sqlite')

        if self.db_type == 'postgresql':
            self.connection_params = {
                'host': config.get('database', 'host'),
                'port': config.getint('database', 'port'),
                'database': config.get('database', 'database'),
                'user': config.get('database', 'user'),
                'password': config.get('database', 'password')
            }
        else:
            self.db_file = config.get('database_dev', 'file',
fallback='ap_support.db')

    def _get_connection(self):
        if self.db_type == 'postgresql':
            conn = psycopg2.connect(**self.connection_params)
            conn.cursor_factory = RealDictCursor
            return conn
        else:
            import sqlite3
            conn = sqlite3.connect(self.db_file)
            conn.row_factory = sqlite3.Row
            return conn
```

Step 4: Update SQL Queries

Replace SQLite-specific SQL with PostgreSQL-compatible SQL:

Changes needed:

- `?` placeholders → `%s` placeholders (for PostgreSQL)
- `AUTOINCREMENT` → `SERIAL`
- `datetime('now')` → `NOW()` or `CURRENT_TIMESTAMP`
- `LIMIT -1 OFFSET n` → `LIMIT ALL OFFSET n`

Example:

```
# SQLite
cursor.execute("INSERT INTO users (name, created_at) VALUES (?, datetime('now'))",
(name,))

# PostgreSQL
cursor.execute("INSERT INTO users (name, created_at) VALUES (%s, NOW())", (name,))
```

Phase 4: Deployment

Step 1: Update All Workstations

1. **Install psycopg2:**

```
pip install psycopg2-binary
```

2. **Copy updated application code**
3. **Create `config/db_config.ini` with server details**
4. **Test connection**

Step 2: User Training

- Inform users about the change
- Multiple users can now work simultaneously
- Database is centrally backed up
- No more individual database files

Backup Strategy

Daily Automated Backup

Create PowerShell script (`backup_database.ps1`):

```
$date = Get-Date -Format "yyyyMMdd_HHmmss"
$backupPath = "C:\Backups\PostgreSQL\vera_support_db_$date.backup"

& "C:\Program Files\PostgreSQL\16\bin\pg_dump.exe" `
    -h localhost `
    -U vera_app_user `
    -d vera_support_db `
    -F c `
    -f $backupPath

# Keep only last 30 days of backups
Get-ChildItem "C:\Backups\PostgreSQL" -Filter "*.backup" |
    Where-Object {$_.CreationTime -lt (Get-Date).AddDays(-30)} |
    Remove-Item
```

Schedule using Windows Task Scheduler to run daily at midnight.

Monitoring and Maintenance

Weekly Tasks

- Check database size: `SELECT pg_database_size('vera_support_db');`
- Review active connections: `SELECT * FROM pg_stat_activity;`
- Verify backups are running

Monthly Tasks

- Review user permissions
- Analyze slow queries
- Update PostgreSQL if new version available
- Test backup restore process

Performance Tuning

- Monitor query performance using pgAdmin
- Add indexes on frequently queried columns
- Run `VACUUM ANALYZE` monthly

Security Considerations

1. **Password Management:**
    - Use strong passwords (min 16 characters)
    - Store config file securely
    - Consider using Windows credential manager
2. **Network Security:**
    - Limit PostgreSQL access to internal network only
    - Use VPN for remote access
    - Enable SSL/TLS for connections (optional but recommended)
3. **User Permissions:**
    - Create read-only users if needed
    - Audit user access regularly
    - Log all database changes

Troubleshooting

Cannot Connect to PostgreSQL

Check:

1. PostgreSQL service is running
2. Firewall allows port 5432
3. `postgresql.conf` has `listen_addresses = '*'`
4. `pg_hba.conf` has correct entry

5. Correct IP address in config file

Test connection:

```
Test-NetConnection -ComputerName server-ip -Port 5432
```

Slow Performance

Solutions:

1. Add indexes on frequently queried columns
2. Run `VACUUM ANALYZE`
3. Increase shared_buffers in postgresql.conf
4. Check for long-running queries

Database Locked Error

This should not occur with PostgreSQL (unlike SQLite). If you see locking issues:

1. Check for zombie connections
2. Review transaction handling in code
3. Ensure proper commit/rollback

Cost Estimate

One-Time Costs

- **Server/VM:** $0 (if using existing hardware) - $500 (new dedicated server)
- **Development Time:** 16-24 hours
- **Testing:** 8 hours

Ongoing Costs

- **Maintenance:** ~2 hours/month
- **Licensing:** $0 (PostgreSQL is free)
- **Backup Storage:** Minimal (few GB)

Total Estimated Implementation Time: 3-4 days

Success Criteria

After implementation, you should have:

- ☑ 10 users can work simultaneously without conflicts
- ☑ No database corruption issues
- ☑ Fast query performance (<1 second for most queries)
- ☑ Automated daily backups

- ☑ Centralized data management
- ☑ User authentication and logging

Alternative: Quick Start with SQLite (Temporary)

If you need a quick solution while planning PostgreSQL migration:

1. Place database on network share with strict file locking
2. Implement application-level locking
3. **Limit to maximum 3 concurrent users**
4. Plan PostgreSQL migration within 1-2 months

Code for application-level locking:

```
import filelock

lock = filelock.FileLock("database.lock")
with lock:
    # Perform database operations
    pass
```

Next Steps

1. **Week 1:** Server setup and PostgreSQL installation
2. **Week 2:** Database migration and code updates
3. **Week 3:** Testing with 2-3 users
4. **Week 4:** Full deployment to all 10 users

Support and Resources

- **PostgreSQL Documentation:** https://www.postgresql.org/docs/
- **psycopg2 Documentation:** https://www.psycopg.org/docs/
- **pgAdmin Documentation:** https://www.pgadmin.org/docs/

Contact

For questions about this implementation, contact your development team or IT administrator.

Document Version: 1.0
Last Updated: November 17, 2025
Application: VERA Support Tool