Database Concurrency

database/sql Connection Pool

Go's `database/sql` provides built-in connection pooling for all SQL drivers.

How it works:

- Pool of database connections
- Connections acquired on query, released after
- Automatic connection reuse
- Configurable pool size

```
import (
    "database/sql"
    _ "github.com/lib/pq"  // PostgreSQL driver
)

db, err := sql.Open("postgres", "postgresql://user:pass@localhost/mydb")
if err != nil {
    log.Fatal(err)
}
defer db.Close()

// Configure pool
db.SetMaxOpenConns(25)                 // Max total connections
db.SetMaxIdleConns(25)                 // Max idle connections
db.SetConnMaxLifetime(5 * time.Minute) // Max connection lifetime
db.SetConnMaxIdleTime(10 * time.Minute) // Max idle time

// Verify connection
if err := db.Ping(); err != nil {
    log.Fatal(err)
}
```

Connection Pool Configuration

```
// For web server with 100 concurrent requests:
db.SetMaxOpenConns(25)  // Limit to 25 concurrent queries
db.SetMaxIdleConns(25)   // Keep 25 connections warm

// For batch processing:
db.SetMaxOpenConns(runtime.NumCPU() * 2)
db.SetMaxIdleConns(runtime.NumCPU())
```

Rules of thumb:

- **MaxOpenConns** = expected peak concurrent queries
- **MaxIdleConns** = MaxOpenConns (keep all connections warm)

- **Don't exceed database limits** (PostgreSQL default: 100 connections)

Basic Query with Context

```
func getUser(ctx context.Context, db *sql.DB, userID int) (*User, error) {
    query := "SELECT id, email, created_at FROM users WHERE id = $1"

    var user User
    err := db.QueryRowContext(ctx, query, userID).Scan(
        &user.ID,
        &user.Email,
        &user.CreatedAt,
    )

    if err == sql.ErrNoRows {
        return nil, fmt.Errorf("user not found")
    }
    if err != nil {
        return nil, err
    }

    return &user, nil
}

// Usage with timeout:
ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
defer cancel()

user, err := getUser(ctx, db, 123)
```

Context benefits:

- Query cancelled if context times out
- Database resources freed immediately
- Prevents long-running queries from piling up

Parallel Queries

Fetch multiple records concurrently.

```
func getUsersParallel(ctx context.Context, db *sql.DB, userIDs []int) ([]*User,
error) {
    type result struct {
        user *User
        err  error
    }

    results := make(chan result, len(userIDs))

    // Fan-out
    for _, id := range userIDs {
```

```
        go func(userID int) {
            user, err := getUser(ctx, db, userID)
            results <- result{user: user, err: err}
        }(id)
    }

    // Fan-in
    users := make([]*User, 0, len(userIDs))
    for range userIDs {
        r := <-results
        if r.err != nil && r.err != sql.ErrNoRows {
            return nil, r.err
        }
        if r.user != nil {
            users = append(users, r.user)
        }
    }

    return users, nil
}
```

Transactions

Transactions require same connection—**DON'T** use concurrency inside transaction.

```
func transferMoney(ctx context.Context, db *sql.DB, fromID, toID int, amount
float64) error {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer tx.Rollback()  // No-op if committed

    // Debit from account
    _, err = tx.ExecContext(ctx,
        "UPDATE accounts SET balance = balance - $1 WHERE id = $2",
        amount, fromID)
    if err != nil {
        return err
    }

    // Credit to account
    _, err = tx.ExecContext(ctx,
        "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
        amount, toID)
    if err != nil {
        return err
    }

    // Commit transaction
```

```
    return tx.Commit()
}
```

Key points:

- `BeginTx` acquires single connection from pool
- All statements in `tx` use same connection
- Connection released when transaction ends
- **Never share `tx` across goroutines**

Prepared Statements

Reuse compiled query for better performance.

```
func insertUsers(ctx context.Context, db *sql.DB, users []User) error {
    // Prepare statement
    stmt, err := db.PrepareContext(ctx,
        "INSERT INTO users (email, created_at) VALUES ($1, $2)")
    if err != nil {
        return err
    }
    defer stmt.Close()

    // Execute multiple times
    for _, user := range users {
        _, err := stmt.ExecContext(ctx, user.Email, user.CreatedAt)
        if err != nil {
            return err
        }
    }

    return nil
}
```

Prepared statements are connection-specific but thread-safe:

- Statement can be used from multiple goroutines
- Automatically creates prepared statement per connection
- Pool manages per-connection statements

Batch Insert

Use transactions for faster batch inserts.

```
func batchInsertUsers(ctx context.Context, db *sql.DB, users []User) error {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer tx.Rollback()

    stmt, err := tx.PrepareContext(ctx,
```

```
        "INSERT INTO users (email, created_at) VALUES ($1, $2)")
    if err != nil {
        return err
    }
    defer stmt.Close()

    for _, user := range users {
        _, err := stmt.ExecContext(ctx, user.Email, user.CreatedAt)
        if err != nil {
            return err
        }
    }

    return tx.Commit()
}

// Even faster: Multi-row insert
func batchInsertOptimized(ctx context.Context, db *sql.DB, users []User) error {
    if len(users) == 0 {
        return nil
    }

    // Build multi-row insert
    valueStrings := make([]string, 0, len(users))
    valueArgs := make([]interface{}, 0, len(users)*2)

    for i, user := range users {
        valueStrings = append(valueStrings, fmt.Sprintf("($%d, $%d)", i*2+1, i*2+2))
        valueArgs = append(valueArgs, user.Email, user.Created At)
    }

    query := fmt.Sprintf("INSERT INTO users (email, created_at) VALUES %s",
        strings.Join(valueStrings, ","))

    _, err := db.ExecContext(ctx, query, valueArgs...)
    return err
}
```

Worker Pool for Database Operations

Limit concurrent database queries.

```
type DBWorkerPool struct {
    db   *sql.DB
    sem  chan struct{}
    wg   sync.WaitGroup
}

func NewDBWorkerPool(db *sql.DB, workers int) *DBWorkerPool {
    return &DBWorkerPool{
        db:  db,
```

```
        sem: make(chan struct{}, workers),
    }
}

func (pool *DBWorkerPool) Query(ctx context.Context, query string, args
...interface{}) (*sql.Rows, error) {
    pool.sem <- struct{}{}
    defer func() { <-pool.sem }()

    return pool.db.QueryContext(ctx, query, args...)
}

func (pool *DBWorkerPool) ProcessBatch(ctx context.Context, items []Item) error {
    errCh := make(chan error, 1)

    for _, item := range items {
        pool.wg.Add(1)

        go func(i Item) {
            defer pool.wg.Done()

            pool.sem <- struct{}{}
            defer func() { <-pool.sem }()

            if err := pool.processItem(ctx, i); err != nil {
                select {
                case errCh <- err:
                default:
                }
            }
        }(item)
    }

    pool.wg.Wait()

    select {
    case err := <-errCh:
        return err
    default:
        return nil
    }
}

func (pool *DBWorkerPool) processItem(ctx context.Context, item Item) error {
    _, err := pool.db.ExecContext(ctx,
        "INSERT INTO items (name, value) VALUES ($1, $2)",
        item.Name, item.Value)
    return err
}
```

Optimistic Locking

Prevent lost updates without locking rows.

```
type Account struct {
    ID      int
    Balance float64
    Version int  // Optimistic lock version
}

func updateBalance(ctx context.Context, db *sql.DB, accountID int, amount float64)
error {
    maxRetries := 3

    for i := 0; i < maxRetries; i++ {
        // Read current version
        var account Account
        err := db.QueryRowContext(ctx,
            "SELECT id, balance, version FROM accounts WHERE id = $1",
            accountID).Scan(&account.ID, &account.Balance, &account.Version)
        if err != nil {
            return err
        }

        // Update with version check
        result, err := db.ExecContext(ctx,
            "UPDATE accounts SET balance = $1, version = version + 1 WHERE id = $2
AND version = $3",
            account.Balance+amount, accountID, account.Version)
        if err != nil {
            return err
        }

        rows, _ := result.RowsAffected()
        if rows == 1 {
            return nil  // Success
        }

        // Version mismatch, retry
        time.Sleep(time.Duration(i*10) * time.Millisecond)
    }

    return fmt.Errorf("update failed after %d retries", maxRetries)
}
```

Pessimistic Locking (SELECT FOR UPDATE)

Lock rows during transaction.

```go
func transferWithLocking(ctx context.Context, db *sql.DB, fromID, toID int, amount
float64) error {
    tx, err := db.BeginTx(ctx, &sql.TxOptions{
        Isolation: sql.LevelSerializable,
    })
    if err != nil {
        return err
    }
    defer tx.Rollback()

    // Lock rows in consistent order (prevent deadlocks)
    ids := []int{fromID, toID}
    sort.Ints(ids)

    var balances [2]float64
    for i, id := range ids {
        err := tx.QueryRowContext(ctx,
            "SELECT balance FROM accounts WHERE id = $1 FOR UPDATE",
            id).Scan(&balances[i])
        if err != nil {
            return err
        }
    }

    // Verify sufficient funds
    fromIndex := 0
    if ids[1] == fromID {
        fromIndex = 1
    }
    if balances[fromIndex] < amount {
        return fmt.Errorf("insufficient funds")
    }

    // Perform transfer
    _, err = tx.ExecContext(ctx,
        "UPDATE accounts SET balance = balance - $1 WHERE id = $2",
        amount, fromID)
    if err != nil {
        return err
    }

    _, err = tx.ExecContext(ctx,
        "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
        amount, toID)
    if err != nil {
        return err
    }

    return tx.Commit()
}
```

Handling Connection Exhaustion

```
func queryWithRetry(ctx context.Context, db *sql.DB, query string) (*sql.Rows,
error) {
    maxRetries := 3
    backoff := 100 * time.Millisecond

    for i := 0; i < maxRetries; i++ {
        rows, err := db.QueryContext(ctx, query)

        if err == nil {
            return rows, nil
        }

        // Check if connection pool exhausted
        if strings.Contains(err.Error(), "connection pool exhausted") {
            log.Printf("Connection pool exhausted, retry %d/%d", i+1, maxRetries)
            time.Sleep(backoff)
            backoff *= 2
            continue
        }

        return nil, err
    }

    return nil, fmt.Errorf("query failed after %d retries", maxRetries)
}
```

Common Mistakes

Mistake 1: Not Closing Rows

```
// WRONG: Connection leak
rows, _ := db.Query("SELECT * FROM users")
for rows.Next() {
    // Process
}
// Missing rows.Close()!

// Fix:
rows, _ := db.Query("SELECT * FROM users")
defer rows.Close()  // Always close

for rows.Next() {
    // Process
}
if err := rows.Err(); err != nil {
```

```
    return err
}
```

Mistake 2: Sharing Transactions Across Goroutines

```
// WRONG: Transaction is not thread-safe
tx, _ := db.Begin()

go func() {
    tx.Exec("INSERT ...") // RACE!
}()

go func() {
    tx.Exec("INSERT ...") // RACE!
}()

// Fix: Use separate transactions or don't use goroutines in transaction
tx, _ := db.Begin()
tx.Exec("INSERT ...")
tx.Exec("INSERT ...")
tx.Commit()
```

Mistake 3: Too Many Open Connections

```
// WRONG: Exceeds database limit
db.SetMaxOpenConns(1000)  // But Postgres allows only 100!

// Fix: Match database limit
db.SetMaxOpenConns(90)  // Leave some for admin connections
```

Mistake 4: Long-Running Transactions

```
// WRONG: Holds connection for entire HTTP request
tx, _ := db.Begin()
defer tx.Commit()

// Expensive computation (holds connection)
result := expensiveComputation()

tx.Exec("INSERT ...", result)

// Fix: Keep transactions short
result := expensiveComputation()

tx, _ := db.Begin()
tx.Exec("INSERT ...", result)
tx.Commit()
```

Performance Tips

1. **Use connection pooling:** Set MaxOpenConns appropriately
2. **Batch inserts:** Use transactions + multi-row INSERT
3. **Prepared statements:** Reuse for repeated queries
4. **Query with context:** Prevents abandoned queries
5. **Close resources:** Always defer `rows.Close()`
6. **Limit parallelism:** Don't spawn more goroutines than connections
7. **Optimize queries:** Use EXPLAIN, add indexes

Interview Questions

Q: "How does database/sql connection pooling work?"

"sql.DB maintains pool of connections. Acquiring connection: call Query/Exec, blocks if pool full unless MaxOpenConns reached, creates new connection if under limit. Releasing: close rows/statements automatically returns connection to pool. Idle connections kept up to MaxIdleConns, closed after ConnMaxIdleTime. Thread-safe, goroutines share same sql.DB instance."

Q: "Can you use transactions across multiple goroutines?"

"No. sql.Tx is not thread-safe, tied to single connection. All operations in transaction must run on same connection sequentially. Sharing tx across goroutines causes races and breaks transaction guarantees. Solution: keep all transaction logic in single goroutine, use channels to pass results."

Q: "What's the difference between optimistic and pessimistic locking?"

"Optimistic: Assume no conflicts, use version field, retry on conflict. No locks, better concurrency, can fail if high contention. Pessimistic: Use SELECT FOR UPDATE to lock rows, guaranteed no conflicts, lower concurrency, risk of deadlocks. Choose optimistic for low contention (rare updates), pessimistic for critical sections (banking)."

Q: "How do you prevent connection pool exhaustion?"

"Set appropriate MaxOpenConns (don't exceed DB limit). Use timeouts on queries (context.WithTimeout). Monitor pool metrics (db.Stats()). Add semaphore to limit concurrent queries. Ensure resources closed (defer rows.Close()). For spikes: increase pool size or add queueing/backpressure. Alert when Stats().OpenConnections approaches MaxOpenConns."

Key Takeaways

1. **sql.DB is connection pool, thread-safe, reuse across goroutines**
2. **sql.Tx is NOT thread-safe, use in single goroutine**
3. **Always use context for timeouts**
4. **Always defer rows.Close()**
5. **Set MaxOpenConns to match database limits**
6. **Batch inserts with transactions**
7. **Prepared statements for repeated queries**
8. **Optimistic locking for low contention**
9. **Pessimistic locking for critical sections**
10. **Monitor connection pool metrics**

Exercises

1. Build connection pool wrapper with metrics (active, idle, wait time).
2. Implement retry logic for connection pool exhaustion.
3. Create batch insert function that handles 10,000+ records efficiently.
4. Write optimistic locking example, test with concurrent updates.
5. Benchmark: Compare single vs. batched vs. parallel inserts.

Next: file-io.md - Concurrent file reading and writing.