The Neo4j Go Driver Manual v5.0

Table of Contents

Quickstart . . . 1
- Installation . . . 1
- Connect to the database . . . 1
- Query the database . . . 2
- Run your own transactions . . . 2
- Close connections and sessions . . . 4
- API documentation . . . 4

Glossary . . . 5

=Basic workflow= . . . 7
- Installation . . . 8
  - Install the driver . . . 8
  - Get a Neo4j instance . . . 8
- Connection . . . 9
  - Connect to the database . . . 9
  - Connect to an Aura instance . . . 10
  - Further connection parameters . . . 10
- Query the database . . . 11
  - Write to the database . . . 11
  - Read from the database . . . 11
  - Update the database . . . 12
  - Delete from the database . . . 13
  - Query parameters . . . 13
  - Query configuration . . . 14
  - A full example . . . 15

=Advanced usage= . . . 17
- Run your own transactions . . . 18
  - Create a session . . . 18
  - Run a managed transaction . . . 18
  - Run an explicit transaction . . . 22
  - Process query results . . . 24
  - Session configuration . . . 25
  - Transaction configuration . . . 26
  - Close sessions . . . 26
- Explore the query execution summary . . . 27
  - Retrieve the execution summary . . . 27
  - Query counters . . . 27
  - Query execution plan . . . 28
  - Notifications . . . 29

Coordinate parallel transactions . . . 31
Bookmarks with ExecuteQuery() . . . 31
Bookmarks within a single session . . . 31
Bookmarks across multiple sessions . . . 32
Mix ExecuteQuery() and sessions . . . 34
Run concurrent transactions . . . 35
Concurrent processing of a query result set (using sessions) . . . 35
Concurrent run of multiple queries (using ExecuteQuery()) . . . 36
Further query mechanisms . . . 38
Implicit (or auto-commit) transactions . . . 38
Dynamic values in property keys, relationship types, and labels . . . 39
Logging . . . 40
Performance recommendations . . . 42
Always specify the target database . . . 42
Be aware of the cost of transactions . . . 42
Don't fetch large result sets all at once . . . 43
Route read queries to cluster readers . . . 46
Create indexes . . . 47
Profile queries . . . 47
Specify node labels . . . 49
Batch data creation . . . 49
Use query parameters . . . 50
Concurrency . . . 50
Use MERGE for creation only when needed . . . 51
Filter notifications . . . 51
=Reference= . . . 52
Advanced connection information . . . 53
Connection URI . . . 53
Connection protocols and security . . . 53
Authentication methods . . . 53
Custom address resolver . . . 54
Further connection parameters . . . 55
Data types and mapping to Cypher types . . . 56
Core types . . . 56
Temporal types . . . 56
Spatial types . . . 58
Graph types . . . 60
Exceptions . . . 62
API documentation . . . 64
=GraphAcademy courses= . . . 65
Graph Data Modeling Fundamentals . . . 66

Intermediate Cypher Queries 67
Building Neo4j Applications with Go 68

Quickstart

The Neo4j Go driver is the official library to interact with a Neo4j instance through a Go application.

At the hearth of Neo4j lies Cypher, the query language to interact with a Neo4j database. While this guide does not *require* you to be a seasoned Cypher querier, it is going to be easier to focus on the Go-specific bits if you already know some Cypher. For this reason, although this guide does *also* provide a gentle introduction to Cypher along the way, consider checking out Getting started → Cypher for a more detailed walkthrough of graph databases modelling and querying if this is your first approach. You may then apply that knowledge while following this guide to develop your Go application.

Installation

From within a module, use `go get` to install the Neo4j Go Driver:

```
go get github.com/neo4j/neo4j-go-driver/v5
```

More info on installing the driver →

Connect to the database

Connect to a database by creating a DriverWithContext object and providing a URL and an authentication token. Once you have a `DriverWithContext` instance, use the `.VerifyConnectivity()` method to ensure that a working connection can be established.

```
package main

import (
    "fmt"
    "context"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()
    // URI examples: "neo4j://localhost", "neo4j+s://xxx.databases.neo4j.io"
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    defer driver.Close(ctx)

    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }
    fmt.Println("Connection established.")
}
```

More info on connecting to a database →

Query the database

Execute a Cypher statement with the function `ExecuteQuery()`. Do not hardcode or concatenate parameters: use placeholders and specify the parameters as keyword arguments.

```
// Get the name of all 42 year-olds
result, _ := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person {age: $age}) RETURN p.name AS name",
    map[string]any{
        "age": "42",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))

// Loop through results and do something with them
for _, record := range result.Records {
    fmt.Println(record.AsMap())
}

// Summary information
fmt.Printf("The query `%v` returned %v records in %+v.\n",
    result.Summary.Query().Text(), len(result.Records),
    result.Summary.ResultAvailableAfter())
```

More info on querying the database →

Run your own transactions

For more advanced use-cases, you can run transactions. Use the methods `Session.ExecuteRead()` and `Session.ExecuteWrite()` to run managed transactions.

A transaction with multiple queries, client logic, and potential roll backs

```
package main

import (
    "fmt"
    "context"
    "strconv"
    "errors"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()
    var employeeThreshold int64 = 10  // Neo4j's integer maps to Go's int64

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)
    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }

    session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
    defer session.Close(ctx)

    // Create 100 people and assign them to various organizations
```

```go
    for i := 0; i < 100; i++ {
        name := "Thor" + strconv.Itoa(i)
        orgId, err := session.ExecuteWrite(ctx,
            func(tx neo4j.ManagedTransaction) (any, error) {
                var orgId string

                // Create new Person node with given name, if not exists already
                _, err := tx.Run(
                    ctx,
                    "MERGE (p:Person {name: $name})",
                    map[string]any{
                        "name": name,
                    })
                if err != nil {
                    return nil, err
                }

                // Obtain most recent organization ID and the number of people linked to it
                result, err := tx.Run(
                    ctx, `
                    MATCH (o:Organization)
                    RETURN o.id AS id, COUNT{(p:Person)-[r:WORKS_FOR]->(o)} AS employeesN
                    ORDER BY o.createdDate DESC
                    LIMIT 1
                    `, nil)
                if err != nil {
                    return nil, err
                }
                org, err := result.Single(ctx)

                // If no organization exists, create one and add Person to it
                if org == nil {
                    orgId, _ = createOrganization(ctx, tx)
                    fmt.Println("No orgs available, created", orgId)
                    err = addPersonToOrganization(ctx, tx, name, orgId)
                    if err != nil {
                        return nil, errors.New("Failed to add person to new org")
                        // Transaction will roll back
                        // -> not even Person and/or Organization is created!
                    }
                } else {
                    orgId = org.AsMap()["id"].(string)
                    if employeesN := org.AsMap()["employeesN"].(int64);
                       employeesN == 0 {
                        return nil, errors.New("Most recent organization is empty")
                        // Transaction will roll back
                        // -> not even Person is created!
                    }

                    // If org does not have too many employees, add this Person to it
                    if employeesN := org.AsMap()["employeesN"].(int64);
                       employeesN < employeeThreshold {
                        err = addPersonToOrganization(ctx, tx, name, orgId)
                        if err != nil {
                            return nil, err
                            // Transaction will roll back
                            // -> not even Person is created!
                        }
                    // Otherwise, create a new Organization and link Person to it
                    } else {
                        orgId, err = createOrganization(ctx, tx)
                        if err != nil {
                            return nil, err
                            // Transaction will roll back
                            // -> not even Person is created!
                        }
                        fmt.Println("Latest org is full, created", orgId)
                        err = addPersonToOrganization(ctx, tx, name, orgId)
                        if err != nil {
                            return nil, err
                            // Transaction will roll back
                            // -> not even Person and/or Organization is created!
                        }
                    }
                }
                // Return the Organization ID to which the new Person ends up in
                return orgId, nil
```

```
            })
        if err != nil {
            fmt.Println(err)
        } else {
            fmt.Println("User", name, "added to organization", orgId)
        }
    }
}

func createOrganization(ctx context.Context, tx neo4j.ManagedTransaction) (string, error) {
    result, err := tx.Run(
        ctx, `
        CREATE (o:Organization {id: randomuuid(), createdDate: datetime()})
        RETURN o.id AS id
        `, nil)
    if err != nil {
        return "", err
    }
    org, err := result.Single(ctx)
    if err != nil {
        return "", err
    }
    orgId, _ := org.AsMap()["id"]
    return orgId.(string), err
}

func addPersonToOrganization(ctx context.Context, tx neo4j.ManagedTransaction, personName string, orgId
string) (error) {
    _, err := tx.Run(
        ctx, `
        MATCH (o:Organization {id: $orgId})
        MATCH (p:Person {name: $name})
        MERGE (p)-[:WORKS_FOR]->(o)
        `, map[string]any{
            "orgId": orgId,
            "name": personName,
        })
    return err
}
```

More info on running transactions →

Close connections and sessions

Call the `.close()` method on all `DriverWithContext` and `SessionWithContext` instances to release any resources still held by them. The best practice is to call the methods with the `defer` keyword as soon as you create new objects.

```
driver, err := neo4j.NewDriverWithContext(dbUri, neo4j.BasicAuth(dbUser, dbPassword, ""))
defer driver.Close(ctx)
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
```

API documentation

For in-depth information about driver features, check out the API documentation.

Glossary

LTS

A *Long Term Support* release is one guaranteed to be supported for a number of years. Neo4j 4.4 is LTS, and Neo4j 5 will also have an LTS version.

Aura

Aura is Neo4j's fully managed cloud service. It comes with both free and paid plans.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the database. It is like SQL, but for graphs.

APOC

Awesome Procedures On Cypher (APOC) is a library of (many) functions that can not be easily expressed in Cypher itself.

Bolt

Bolt is the protocol used for interaction between Neo4j instances and drivers. It listens on port 7687 by default.

ACID

Atomicity, Consistency, Isolation, Durability (ACID) are properties guaranteeing that database transactions are processed reliably. An ACID-compliant DBMS ensures that the data in the database remains accurate and consistent despite failures.

eventual consistency

A database is eventually consistent if it provides the guarantee that all cluster members will, *at some point in time*, store the latest version of the data.

causal consistency

A database is causally consistent if read and write queries are seen by every member of the cluster in the same order. This is stronger than *eventual consistency*.

NULL

The null marker is not a type but a placeholder for absence of value. For more information, see Cypher → Working with `null`.

transaction

A transaction is a unit of work that is either *committed* in its entirety or *rolled back* on failure. An example is a bank transfer: it involves multiple steps, but they must *all* succeed or be reverted, to avoid money being subtracted from one account but not added to the other.

backpressure

Backpressure is a force opposing the flow of data. It ensures that the client is not being overwhelmed by data faster than it can handle.

transaction function

A transaction function is a callback executed by an `ExecuteRead` or `ExecuteWrite` call. The driver automatically re-executes the callback in case of server failure.

DriverWithContext

A `DriverWithContext` object holds the details required to establish connections with a Neo4j database.

=Basic workflow=

Installation

To start creating a Neo4j Go application, you first need to install the Go Driver and get a Neo4j database instance to connect to.

Install the driver

If you are starting from scratch, the first step is to initialize a Go module. You can do so by creating a directory, entering it, and using `go mod init`:

```
mkdir neo4j-app
cd neo4j-app
go mod init neo4j-app
```

From within a module, use `go get` to install the Neo4j Go Driver:

```
go get github.com/neo4j/neo4j-go-driver/v5
```

Always use the latest version of the driver, as it will always work both with the previous Neo4j LTS release and with the current and next major releases. The latest `5.x` driver supports connection to any Neo4j 5 and 4.4 instance, and will also be compatible with Neo4j 6. For a detailed list of changes across versions, see the driver's changelog.

The Neo4j Go Driver is compatible (and requires) any officially maintained Go version.

Get a Neo4j instance

You need a running Neo4j database in order to use the driver with it. The easiest way to spin up a **local instance** is through a Docker container (requires `docker.io`). The command below runs the latest Neo4j version in Docker, setting the admin username to `neo4j` and password to `secretgraph`:

```
docker run \
   -p7474:7474 \                          # forward port 7474 (HTTP)
   -p7687:7687 \                          # forward port 7687 (Bolt)
   -d \                                   # run in background
   -e NEO4J_AUTH=neo4j/secretgraph \      # set login credentials
   neo4j:latest
```

Alternatively, you can obtain a free **cloud instance** through Aura.

You can also install Neo4j on your system, or use Neo4j Desktop to create a local development environment (not for production).

Connection

Once you have installed the driver and have a running Neo4j instance, you are ready to connect your application to the database.

Connect to the database

You connect to a Neo4j database by creating a DriverWithContext object and providing a URL and an authentication token.

```
package main

import (
    "context"
    "fmt"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background() ②
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(  ①
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)  ④

    err = driver.VerifyConnectivity(ctx)  ③
    if err != nil {
        panic(err)
    }
    fmt.Println("Connection established.")
}
```

① Creating a `DriverWithContext` instance only provides information on *how* to access the database, but does not actually *establish* a connection. Connection is instead deferred to when the first query is executed.

② Most driver functions require a `context.Context` parameter, see the context package.

③ To verify immediately that the driver can connect to the database (valid credentials, compatible versions, etc), use the `.VerifyConnectivity(ctx)` method after initializing the driver.

④ Always close `DriverWithContext` objects to free up all allocated resources, even upon unsuccessful connection or runtime errors in subsequent querying. The safest practice is to `defer` the call to `DriverWithContext.Close(ctx)` after the object is successfully created. Note that there are corner cases in which `.Close()` might return an error, so you may want to catch that as well.

`Driver` objects are immutable, thread-safe, and expensive to create, so your application should create only one instance and pass it around (you may share `Driver` instances across threads). If you need to query the database through several different users, use impersonation without creating a new `DriverWithContext` instance. If you want to alter a `DriverWithContext` configuration, you need to create a new object.

Connect to an Aura instance

When you create an Aura instance, you may download a text file (a so-called *Dotenv file*) containing the connection information to the database in the form of environment variables. The file has a name of the form `Neo4j-a0a2fa1d-Created-2023-11-06.txt`.

You can either manually extract the URI and the credentials from that file, or use a third party-module to load them. We recommend the module package `godotenv` for that purpose.

```
package main

import (
    "context"
    "os"
    "fmt"
    "github.com/joho/godotenv"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()
    err := godotenv.Load("Neo4j-a0a2fa1d-Created-2023-11-06.txt")
    if err != nil {
        panic(err)
    }
    dbUri := os.Getenv("NEO4J_URI")
    dbUser := os.Getenv("NEO4J_USERNAME")
    dbPassword := os.Getenv("NEO4J_PASSWORD")
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)

    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }
    fmt.Println("Connection established.")
}
```

An Aura instance is not conceptually different from any other Neo4j instance, as Aura is simply a *deployment mode* for Neo4j. When interacting with a Neo4j database through the driver, it doesn't make a difference whether it is an Aura instance it is working with or a different deployment.

Further connection parameters

For more `DriverWithContext` configuration parameters and further connection settings, see Advanced connection information.

Query the database

Once you have connected to the database, you can run queries using Cypher and the function `ExecuteQuery()`.

`ExecuteQuery()` was introduced with the version 5.8 of the driver.
For queries with earlier versions, use sessions and transactions.

Write to the database

To create a node representing a person named `Alice`, use the Cypher clause `CREATE`:

Create a node representing a person named `Alice`

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "CREATE (p:Person {name: $name}) RETURN p",  ①
    map[string]any{  ②
        "name": "Alice",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))  ③
if err != nil {
    panic(err)
}

summary := result.Summary  ④
fmt.Printf("Created %v nodes in %+v.\n",
    summary.Counters().NodesCreated(),
    summary.ResultAvailableAfter())
```

① The Cypher query

② A map of *query parameters*

③ Which database the query should be run against

④ The summary of execution returned by the server

Read from the database

To retrieve information from the database, use the Cypher clause `MATCH`:

Retrieve all `Person` nodes

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person) RETURN p.name AS name",
    nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
if err != nil {
    panic(err)
}

// Loop through results and do something with them
for _, record := range result.Records { ①
    name, _ := record.Get("name")  // .Get() 2nd return is whether key is present
    fmt.Println(name)
    // or
    // fmt.Println(record.AsMap())  // get Record as a map
}

// Summary information  ②
fmt.Printf("The query `%v` returned %v records in %+v.\n",
    result.Summary.Query().Text(), len(result.Records),
    result.Summary.ResultAvailableAfter())
```

① `result.Records` contains the result as an array of `Record` objects

② `result.Summary` contains the summary of execution returned by the server

When accessing a record's content, all its properties are of type any. This means that you have to cast them to the relevant Go type if you want to use methods/features defined on such types. For example, if the `name` property coming from the database is a string, `record.AsMap()["name"][1]` would result in an *invalid operation* error at compilation time. For it to work, cast the value to string *before* using it as a string: `name := record.AsMap()["name"].(string)` and then `name[1]`.

Update the database

To update a node's information in the database, use the Cypher clauses `SET`:

Update node `Alice` to add an `age` property

```
result, err := neo4j.ExecuteQuery(ctx, driver, `
    MATCH (p:Person {name: $name})
    SET p.age = $age
    `, map[string]any{
        "name": "Alice",
        "age": 42,
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
if err != nil {
    panic(err)
}
fmt.Println("Query updated the database?",
    result.Summary.Counters().ContainsUpdates())
```

To create a new relationship, linking it to two already existing node, use a combination of the Cypher clauses `MATCH` and `CREATE`:

Create a relationship `:KNOWS` between `Alice` and `Bob`

```
result, err := neo4j.ExecuteQuery(ctx, driver, `
    MATCH (alice:Person {name: $name})  ①
    MATCH (bob:Person {name: $friend})  ②
    CREATE (alice)-[:KNOWS]->(bob)  ③
    `, map[string]any{
        "name": "Alice",
        "friend": "Bob",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
if err != nil {
    panic(err)
}
fmt.Println("Query updated the database?",
    result.Summary.Counters().ContainsUpdates())
```

① Retrieve the person node named `Alice` and bind it to a variable `alice`

② Retrieve the person node named `Bob` and bind it to a variable `bob`

③ Create a new `:KNOWS` relationship outgoing from the node bound to `alice` and attach to it the `Person` node named `Bob`

Delete from the database

To remove a node and any relationship attached to it, use the Cypher clause `DETACH DELETE`:

Remove the `Alice` node and all its relationships

```
// This does not delete _only_ p, but also all its relationships!
result, err := neo4j.ExecuteQuery(ctx, driver, `
    MATCH (p:Person {name: $name})
    DETACH DELETE p
    `, map[string]any{
        "name": "Alice",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
if err != nil {
    panic(err)
}
fmt.Println("Query updated the database?",
    result.Summary.Counters().ContainsUpdates())
```

Query parameters

Do not hardcode or concatenate parameters directly into queries. Instead, always use placeholders and specify the Cypher parameters, as shown in the previous examples. This is for:

1. **performance benefits**: Neo4j compiles and caches queries, but can only do so if the query structure is unchanged;
2. **security reasons**: see Protecting against Cypher Injection.

Query parameters should get grouped into a map and passed as second parameter to `ExecuteQuery()`. If a query has no parameters, you can pass `nil` instead of an empty map.

```go
parameters := map[string]any{
    "name": "Alice",
    "age": 42,
}
neo4j.ExecuteQuery(ctx, driver,
    "MERGE (:Person {name: $name, age: $age})",
    parameters,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

There can be circumstances where your query structure prevents the usage of parameters in all its parts. For those rare use cases, see Dynamic values in property keys, relationship types, and labels.

Query configuration

You can supply further configuration parameters to alter the default behavior of `ExecuteQuery()`. These are provided as an arbitrary number of callbacks from the 4th function argument onward.

Database selection

It is recommended to **always specify the database explicitly** with the `neo4j.ExecuteQueryWithDatabase("<dbName>")` callback, even on single-database instances. This allows the driver to work more efficiently, as it saves a network round-trip to the server to resolve the home database. If no database is given, the user's home database is used.

```go
neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person) RETURN p.name",
    nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

Specifying the database through the configuration method is preferred over the `USE` Cypher clause. If the server runs on a cluster, queries with `USE` require server-side routing to be enabled. Queries may also take longer to execute as they may not reach the right cluster member at the first attempt, and need to be routed to one containing the requested database.

Request routing

In a cluster environment, all queries are directed to the leader node by default. To improve performance on read queries, you can use the callback `neo4j.ExecuteQueryWithReadersRouting()` to route a query to the read nodes.

```go
neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person) RETURN p.name",
    nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"),
    neo4j.ExecuteQueryWithReadersRouting())
```

Although executing a *write* query in read mode likely results in a runtime error, **you should not rely on this for access control.** The difference between the two modes is that *read* transactions will be routed to any node of a cluster, whereas *write* ones will be directed to the leader. In other words, there is no guarantee that a write query submitted in read mode will be rejected.

Run queries as a different user

You can execute a query under the security context of a different user with the callback `neo4j.ExecuteQueryWithImpersonatedUser("<somebodyElse>")`, specifying the name of the user to impersonate. For this to work, the user under which the `DriverWithContext` object was created needs to have the appropriate permissions. Impersonating a user is cheaper than creating a new `DriverWithContext` object.

```
neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person) RETURN p.name",
    nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"),
    neo4j.ExecuteQueryWithImpersonatedUser("<somebodyElse>"))
```

When impersonating a user, the query is run within the complete security context of the impersonated user and not the authenticated user (i.e. home database, permissions, etc.).

A full example

```
package main

import (
    "fmt"
    "context"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)
    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }

    // Prepare data
    people := []map[string]any {
       {"name": "Alice", "age": 42, "friends": []string{"Bob", "Peter", "Anna"},},
       {"name": "Bob", "age": 19,},
       {"name": "Peter", "age": 50,},
       {"name": "Anna", "age": 30,},
    }
```

```go
    // Create some nodes
    for _, person := range people {
        _, err := neo4j.ExecuteQuery(ctx, driver,
            "MERGE (p:Person {name: $person.name, age: $person.age})",
            map[string]any{
                "person": person,
            }, neo4j.EagerResultTransformer,
            neo4j.ExecuteQueryWithDatabase("neo4j"))
        if err != nil {
            panic(err)
        }
    }

    // Create some relationships
    for _, person := range people {
        if person["friends"] != "" {
            _, err := neo4j.ExecuteQuery(ctx, driver, `
                MATCH (p:Person {name: $person.name})
                UNWIND $person.friends AS friend_name
                MATCH (friend:Person {name: friend_name})
                MERGE (p)-[:KNOWS]->(friend)
                `, map[string]any{
                    "person": person,
                }, neo4j.EagerResultTransformer,
                neo4j.ExecuteQueryWithDatabase("neo4j"))
            if err != nil {
                panic(err)
            }
        }
    }

    // Retrieve Alice's friends who are under 40
    result, err := neo4j.ExecuteQuery(ctx, driver, `
        MATCH (p:Person {name: $name})-[:KNOWS]-(friend:Person)
        WHERE friend.age < $age
        RETURN friend
        `, map[string]any{
            "name": "Alice",
            "age": 40,
        }, neo4j.EagerResultTransformer,
        neo4j.ExecuteQueryWithDatabase("neo4j"))
    if err != nil {
        panic(err)
    }

    // Loop through results and do something with them
    for _, record := range result.Records {
        person, _ := record.Get("friend")
        fmt.Println(person)
        // or
        // fmt.Println(record.AsMap())
    }

    // Summary information
    fmt.Printf("\nThe query `%v` returned %v records in %+v.\n",
        result.Summary.Query().Text(), len(result.Records),
        result.Summary.ResultAvailableAfter())
}
```

For more information see API documentation → ExecuteQuery().

=Advanced usage=

Run your own transactions

When querying the database with `ExecuteQuery()`, the driver automatically creates a *transaction*. A transaction is a unit of work that is either *committed* in its entirety or *rolled back* on failure. You can include multiple Cypher statements in a single query, as for example when using `MATCH` and `CREATE` in sequence to update the database, but you cannot have multiple queries and interleave some client-logic in between them.

For these more advanced use-cases, the driver provides functions to take full control over the transaction lifecycle. These are called *managed transactions*, and you can think of them as a way of unwrapping the flow of `executableQuery()` and being able to specify its desired behavior in more places.

Create a session

Before running a transaction, you need to obtain a *session*. Sessions act as concrete query channels between the driver and the server, and ensure causal consistency is enforced.

Sessions are created with the method `DriverWithContext.NewSession()`. Use the second argument to alter the session's configuration, among which for example the target database. For further configuration parameters, see Session configuration.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
```

Session creation is a lightweight operation, so sessions can be created and destroyed without significant cost. Always close sessions when you are done with them.

Sessions are *not* thread safe: you can share the main `DriverWithContext` object across threads, but make sure each routine creates its own sessions.

Run a managed transaction

A transaction can contain any number of queries. As Neo4j is ACID compliant, **queries within a transaction will either be executed as a whole or not at all**: you cannot get a part of the transaction succeeding and another failing. Use transactions to group together related queries which work together to achieve a single *logical* database operation.

A managed transaction is created with the methods `SessionWithContext.ExecuteRead()` and `SessionWithContext.ExecuteWrite()`, depending on whether you want to retrieve data from the database or alter it. Both methods take a transaction function callback, which is responsible for actually carrying out the queries and processing the result.

Retrieve people whose name starts with `Al`

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})  ①
defer session.Close(ctx)
people, err := session.ExecuteRead(ctx,  ②
    func(tx neo4j.ManagedTransaction) (any, error) {  ③
        result, err := tx.Run(ctx, `  ④
            MATCH (p:Person) WHERE p.name STARTS WITH $filter
            RETURN p.name AS name ORDER BY name
            `, map[string]any{
                "filter": "Al",
            })
        if err != nil {
            return nil, err
        }
        records, err := result.Collect(ctx)  ⑤
        if err != nil {
            return nil, err
        }
        return records, nil
    })
for _, person := range people.([]*neo4j.Record) {
    fmt.Println(person.AsMap())
}
```

① Create a session. A single session can be the container for multiple queries. Remember to close it when done (here we `defer` its closure just after opening).

② The `.ExecuteRead()` (or `.ExecuteWrite()`) method is the entry point into a transaction.

③ The transaction function callback is responsible of running queries.

④ Use the method `ManagedTransaction.Run()` to run queries. Each query run returns a `ResultWithContext` object.

⑤ Process the result using any of the methods on `ResultWithContext`. The method `.Collect()` retrieves all records into a list.

Do not hardcode or concatenate parameters directly into the query. Use query parameters instead, both for performance and security reasons.

Transaction functions should never return the result object directly. Instead, always process the result in some way. Within a transaction function, a `return` statement where `error` is `nil` results in the transaction being committed, while the transaction is automatically rolled back if the returned `error` value is not `nil`.

The methods `.ExecuteRead()` and `.ExecuteWrite()` have replaced `.ReadTransaction()` and `.WriteTransaction()`, which are deprecated in version 5.x and will be removed in version 6.0.

A transaction with multiple queries, client logic, and potential roll backs

```
package main

import (
    "fmt"
    "context"
    "strconv"
    "errors"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()
    var employeeThreshold int64 = 10  // Neo4j's integer maps to Go's int64
```

```
	// Connection to database
	dbUri := "<URI for Neo4j database>"
	dbUser := "<Username>"
	dbPassword := "<Password>"
	driver, err := neo4j.NewDriverWithContext(
		dbUri,
		neo4j.BasicAuth(dbUser, dbPassword, ""))
	if err != nil {
		panic(err)
	}
	defer driver.Close(ctx)
	err = driver.VerifyConnectivity(ctx)
	if err != nil {
		panic(err)
	}

	session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
	defer session.Close(ctx)

	// Create 100 people and assign them to various organizations
	for i := 0; i < 100; i++ {
		name := "Thor" + strconv.Itoa(i)
		orgId, err := session.ExecuteWrite(ctx,
			func(tx neo4j.ManagedTransaction) (any, error) {
				var orgId string

				// Create new Person node with given name, if not exists already
				_, err := tx.Run(
					ctx,
					"MERGE (p:Person {name: $name})",
					map[string]any{
						"name": name,
					})
				if err != nil {
					return nil, err
				}

				// Obtain most recent organization ID and the number of people linked to it
				result, err := tx.Run(
					ctx, `
					MATCH (o:Organization)
					RETURN o.id AS id, COUNT{(p:Person)-[r:WORKS_FOR]->(o)} AS employeesN
					ORDER BY o.createdDate DESC
					LIMIT 1
					`, nil)
				if err != nil {
					return nil, err
				}
				org, err := result.Single(ctx)

				// If no organization exists, create one and add Person to it
				if org == nil {
					orgId, _ = createOrganization(ctx, tx)
					fmt.Println("No orgs available, created", orgId)
					err = addPersonToOrganization(ctx, tx, name, orgId)
					if err != nil {
						return nil, errors.New("Failed to add person to new org")
						// Transaction will roll back
						// -> not even Person and/or Organization is created!
					}
				} else {
					orgId = org.AsMap()["id"].(string)
					if employeesN := org.AsMap()["employeesN"].(int64);
					   employeesN == 0 {
						return nil, errors.New("Most recent organization is empty")
						// Transaction will roll back
						// -> not even Person is created!
					}

					// If org does not have too many employees, add this Person to it
					if employeesN := org.AsMap()["employeesN"].(int64);
					   employeesN < employeeThreshold {
						err = addPersonToOrganization(ctx, tx, name, orgId)
						if err != nil {
							return nil, err
							// Transaction will roll back
							// -> not even Person is created!
```

```
                    }
                // Otherwise, create a new Organization and link Person to it
                } else {
                    orgId, err = createOrganization(ctx, tx)
                    if err != nil {
                        return nil, err
                        // Transaction will roll back
                        // -> not even Person is created!
                    }
                    fmt.Println("Latest org is full, created", orgId)
                    err = addPersonToOrganization(ctx, tx, name, orgId)
                    if err != nil {
                        return nil, err
                        // Transaction will roll back
                        // -> not even Person and/or Organization is created!
                    }
                }
            }
            // Return the Organization ID to which the new Person ends up in
            return orgId, nil
        })
        if err != nil {
            fmt.Println(err)
        } else {
            fmt.Println("User", name, "added to organization", orgId)
        }
    }
}

func createOrganization(ctx context.Context, tx neo4j.ManagedTransaction) (string, error) {
    result, err := tx.Run(
        ctx, `
        CREATE (o:Organization {id: randomuuid(), createdDate: datetime()})
        RETURN o.id AS id
        `, nil)
    if err != nil {
        return "", err
    }
    org, err := result.Single(ctx)
    if err != nil {
        return "", err
    }
    orgId, _ := org.AsMap()["id"]
    return orgId.(string), err
}

func addPersonToOrganization(ctx context.Context, tx neo4j.ManagedTransaction, personName string, orgId
string) (error) {
    _, err := tx.Run(
        ctx, `
        MATCH (o:Organization {id: $orgId})
        MATCH (p:Person {name: $name})
        MERGE (p)-[:WORKS_FOR]->(o)
        `, map[string]any{
            "orgId": orgId,
            "name": personName,
        })
    return err
}
```

Should a transaction fail for a reason that the driver deems transient, it automatically retries to run the transaction function (with an exponentially increasing delay). For this reason, **transaction functions must be *idempotent*** (i.e., they should produce the same effect when run several times), because you do not know upfront how many times they are going to be executed. In practice, this means that you should not edit nor rely on globals, for example. Note that although transactions functions might be executed multiple times, the queries inside it will always run only once.

A session can chain multiple transactions, but **only one single transaction can be active within a session at any given time.** To maintain multiple concurrent transactions, use multiple concurrent sessions.

Run an explicit transaction

You can achieve full control over transactions by manually beginning one with the method `SessionWithContext.BeginTransaction()`. You run queries inside an explicit transaction with the method `ExplicitTransaction.Run()`.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
tx, err := session.BeginTransaction(ctx)
if err != nil {
    panic(err)
}
    // use tx.Run() to run queries
    //     tx.Commit() to commit the transaction
    //     tx.Rollback() to rollback the transaction
```

An explicit transaction can be committed with `ExplicitTransaction.Commit()` or rolled back with `ExplicitTransaction.Rollback()`. If no explicit action is taken, the driver automatically rolls back the transaction at the end of its lifetime.

Explicit transactions are most useful for applications that need to distribute Cypher execution across multiple functions for the same transaction, or for applications that need to run multiple queries within a single transaction but without the automatic retries provided by managed transactions.

A sketch of an explicit transaction interacting with external APIs

```
package main

import (
    "fmt"
    "context"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)
    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }
    customerId, err := createCustomer(ctx, driver)
    if err != nil {
        panic(err)
    }
    otherBankId := 42
    transferToOtherBank(ctx, driver, customerId, otherBankId, 999)
}

func createCustomer(ctx context.Context, driver neo4j.DriverWithContext) (string, error) {
    result, err := neo4j.ExecuteQuery(ctx, driver, `
    MERGE (c:Customer {id: randomUUID()})
    RETURN c.id AS id
    `, nil,
    neo4j.EagerResultTransformer,
```

```
    neo4j.ExecuteQueryWithDatabase("neo4j"))
    if err != nil {
        return "", err
    }
    customerId, _ := result.Records[0].Get("id")
    return customerId.(string), err
}

func transferToOtherBank(ctx context.Context, driver neo4j.DriverWithContext, customerId string,
otherBankId int, amount float32) {
    session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
    defer session.Close(ctx)
    tx, err := session.BeginTransaction(ctx)
    if err != nil {
        panic(err)
    }

    if ! customerBalanceCheck(ctx, tx, customerId, amount) {
        // give up
        return
    }

    otherBankTransferApi(ctx, customerId, otherBankId, amount)
    // Now the money has been transferred => can't rollback anymore
    // (cannot rollback external services interactions)

    err = decreaseCustomerBalance(ctx, tx, customerId, amount)
    if err != nil {
        requestInspection(ctx, customerId, otherBankId, amount, err)
    }
    err = tx.Commit(ctx)
    if err != nil {
        requestInspection(ctx, customerId, otherBankId, amount, err)
    }
}

func customerBalanceCheck(ctx context.Context, tx neo4j.ExplicitTransaction, customerId string, amount
float32) (bool) {
    result, err := tx.Run(ctx, `
        MATCH (c:Customer {id: $id})
        RETURN c.balance >= $amount AS sufficient
        `, map[string]any{
            "id": customerId,
            "amount": amount,
        })
    if err == nil {
        return false
    }
    record, err := result.Single(ctx)
    if err == nil {
        return false
    }
    sufficient := record.AsMap()["sufficient"]
    return sufficient.(bool)
}

func otherBankTransferApi(ctx context.Context, customerId string, otherBankId int, amount float32) {
    // make some API call to other bank
}

func decreaseCustomerBalance(ctx context.Context, tx neo4j.ExplicitTransaction, customerId string, amount
float32) (error) {
    _, err := tx.Run(ctx, `
        MATCH (c:Customer {id: $id})
        SET c.balance = c.balance - $amount
        `, map[string]any{
            "id": customerId,
            "amount": amount,
        })
    return err
}

func requestInspection(ctx context.Context, customerId string, otherBankId int, amount float32, err error)
{
    // manual cleanup required; log this or similar
    fmt.Println("WARNING: transaction rolled back due to exception:", err)
    fmt.Println("customerId:", customerId, "otherBankId:", otherBankId, "amount:", amount)
```

```
}
```

Process query results

The driver's output of a query is a `ResultWithContext` object, which does not directly contain the result records. Rather, it encapsulates the Cypher result in a rich data structure that requires some parsing on the client side. There are two main points to be aware of:

- **The result records are not immediately and entirely fetched and returned by the server.** Instead, results come as a *lazy stream*. In particular, when the driver receives some records from the server, they are initially *buffered* in a background queue. Records stay in the buffer until they are *consumed* by the application, at which point they are *removed from the buffer*. When no more records are available, the result is *exhausted*.
- **The result acts as a cursor.** This means that there is no way to retrieve a previous record from the stream, unless you saved it in an auxiliary data structure.

The animation below follows the path of a single query: it shows how the driver works with result records and how the application should handle results.

```
<video
  class="rounded-corners"
  controls
  width="100%"
  src="../../../common-content/5/_images/result.mp4"
  poster="../../../common-content/5/_images/result-poster.jpg"
  type="video/mp4"></video>
```

The easiest way of processing a result is by calling `.Collect(ctx)` on it, which yields an array of `Record` objects. Otherwise, a `ResultWithContext` object implements a number of methods for processing records. The most commonly needed ones are listed below.

Name	Description
`Collect(ctx) ([]*Record, error)`	Return the remainder of the result as a list.
`Single(ctx) (*Record, error)`	Return the next and only remaining record, or `nil`. Calling this method always exhausts the result. If more (or less) than one record is available, a non-`nil` error is returned.
`Record() *Record`	Return the current record.
`Next(ctx) bool`	Return `true` if there is a record to be processed after the current one. In that case, it also advances the result iterator.
`Consume(ctx) (ResultSummary, error)`	Return the query result summary. It exhausts the result, so should only be called when data processing is over.

For a complete list of `ResultWithContext` methods, see API documentation → ResultWithContext.

Session configuration

Database selection

It is recommended to **always specify the database explicitly** with the configuration parameter `DatabaseName` upon session creation, even on single-database instances. This allows the driver to work more efficiently, as it saves a network round-trip to the server to resolve the home database. If no database is given, the default database set in the Neo4j instance settings is used.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{
    DatabaseName: "neo4j",
})
```

Specifying the database through the configuration method is preferred over the `USE` Cypher clause. If the server runs on a cluster, queries with `USE` require server-side routing to be enabled. Queries may also take longer to execute as they may not reach the right cluster member at the first attempt, and need to be routed to one containing the requested database.

Request routing

In a cluster environment, all sessions are opened in write mode, routing them to the leader. You can change this by explicitly setting the configuration parameter `AccessMode` to either `neo4j.AccessModeRead` or `neo4j.AccessModeWrite`. Note that `.ExecuteRead()` and `.ExecuteWrite()` automatically override the session's default access mode.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{
    DatabaseName: "neo4j",
    AccessMode: neo4j.AccessModeRead,
})
```

Although executing a *write* query in read mode likely results in a runtime error, **you should not rely on this for access control.** The difference between the two modes is that *read* transactions will be routed to any node of a cluster, whereas *write* ones will be directed to the leader. In other words, there is no guarantee that a write query submitted in read mode will be rejected.

Similar remarks hold for the `.ExecuteRead()` and `.ExecuteWrite()` methods.

Run queries as a different user (impersonation)

You can execute a query under the security context of a different user with the configuration parameter `ImpersonatedUser`, specifying the name of the user to impersonate. For this to work, the user under which the `DriverWithContext` was created needs to have the appropriate permissions. Impersonating a user is cheaper than creating a new `DriverWithContext` object.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{
    DatabaseName: "neo4j",
    ImpersonatedUser: "<somebodyElse>",
})
```

When impersonating a user, the query is run within the complete security context of the impersonated user and not the authenticated user (i.e. home database, permissions, etc.).

Transaction configuration

You can exert further control on transactions by providing configuration callbacks to `.ExecuteRead()`, `.ExecuteWrite()`, and `.BeginTransaction()`. Use them to specify:

- A transaction timeout (in seconds). Transactions that run longer will be terminated by the server. The default value is set on the server side. The minimum value is one millisecond.
- A map of metadata that gets attached to the transaction. These metadata get logged in the server `query.log`, and are visible in the output of the `SHOW TRANSACTIONS` Cypher command. Use this to tag transactions.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
people, err := session.ExecuteRead(ctx,
    func(tx neo4j.ManagedTransaction) (any, error) {
        result, _ := tx.Run(ctx, "MATCH (:Person) RETURN count(*) AS n", nil)
        return result.Collect(ctx)
    },
    neo4j.WithTxTimeout(5*time.Second),  // remember to import `time`
    neo4j.WithTxMetadata(map[string]any{"appName": "peopleTracker"}))
```

Close sessions

Each connection pool has **a finite number of sessions**, so if you open sessions without ever closing them, your application could run out of them. It is thus recommended to call `session.Close()` with the `defer` keyword as soon as you create a new session, to be sure it will be closed in all cases. When a session is closed, it is returned to the connection pool to be later reused.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
// session usage
```

There are corner-cases in which session closure might return an error, so you may want to catch those cases as well.

Explore the query execution summary

After all results coming from a query have been processed, the server ends the transaction by returning a summary of execution. It comes as a `ResultSummary` object, and it contains information among which:

- Query counters — What changes the query triggered on the server
- Query execution plan — How the database would execute (or executed) the query
- Notifications — Extra information raised by the server while running the query
- Timing information and query request summary

Retrieve the execution summary

When running queries with `ExecuteQuery()`, the execution summary is part of the default return object, under the `Summary` key.

```
result, _ := neo4j.ExecuteQuery(ctx, driver, `
    UNWIND ["Alice", "Bob"] AS name
    MERGE (p:Person {name: name})
    `, nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
summary := result.Summary
```

If you are using transaction functions, you can retrieve the query execution summary with the method `Result.Consume()`. **Notice that once you ask for the execution summary, the result stream is exhausted.** This means that any record which has not yet been processed is discarded.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
summary, _ := session.ExecuteWrite(ctx,
    func(tx neo4j.ManagedTransaction) (any, error) {
        result, err := tx.Run(ctx, `
            UNWIND ["Alice", "Bob"] AS name
            MERGE (p:Person {name: name})
            `, nil)
        summary, _ := result.Consume(ctx)
        return summary, err
    })
```

Query counters

The method `ResultSummary.Counters()` returns counters for the operations that a query triggered (as a `Counters` object).

Insert some data and display the query counters

```
result, _ := neo4j.ExecuteQuery(ctx, driver, `
    MERGE (p:Person {name: $name})
    MERGE (p)-[:KNOWS]->(:Person {name: $friend})
    `, map[string]any{
        "name": "Mark",
        "friend": "Bob",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
summary := result.Summary
counters := summary.Counters()
fmt.Println("Nodes created:", counters.NodesCreated())
fmt.Println("Labels added:", counters.LabelsAdded())
fmt.Println("Properties set:", counters.PropertiesSet())
fmt.Println("Relationships created:", counters.RelationshipsCreated())

// Nodes created: 2
// Labels added: 2
// Properties set: 2
// Relationships created: 1
```

There are two additional boolean methods which act as meta-counters:

- `.ContainsUpdates()` — whether the query triggered any write operation on the database on which it ran
- `.ContainsSystemUpdates()` — whether the query updated the `system` database

Query execution plan

If you prefix a query with `EXPLAIN`, the server will return the plan it *would* use to run the query, but will not actually run it. You can retrieve the plan by calling `ResultSummary.Plan()`, which contains the list of Cypher operators that would be used to retrieve the result set. You may use this information to locate potential bottlenecks or room for performance improvements (for example through the creation of indexes).

```
result, _ := neo4j.ExecuteQuery(ctx, driver,
    "EXPLAIN MATCH (p {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    },
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
fmt.Println(result.Summary.Plan().Arguments()["string-representation"])
/*
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+---------------------+
| Operator        | Details        | Estimated Rows | Pipeline            |
+-----------------+----------------+----------------+---------------------+
| +ProduceResults | p              |              1 |                     |
| |               +----------------+----------------+                     |
| +Filter         | p.name = $name |              1 |                     |
| |               +----------------+----------------+                     |
| +AllNodesScan   | p              |             10 | Fused in Pipeline 0 |
+-----------------+----------------+----------------+---------------------+

Total database accesses: ?
*/
```

If you instead prefix a query with the keyword `PROFILE`, the server will return the execution plan it has used

to run the query, together with profiler statistics. This includes the list of operators that were used and additional profiling information about each intermediate step. You can access the plan by calling `ResultSummary.Profile()`. Notice that the query is also *run*, so the result object also contains any result records.

```
result, _ := neo4j.ExecuteQuery(ctx, driver,
    "PROFILE MATCH (p {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    },
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
fmt.Println(result.Summary.Profile().Arguments()["string-representation"])
/*
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+
| Operator        | Details        | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache
Hits/Misses | Time (ms) | Pipeline            |
+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+
| +ProduceResults | p              |              1 |    1 |       3 |                |
|           |                      |
| |               +----------------+----------------+------+---------+----------------+
|           |                      |
| +Filter         | p.name = $name |              1 |    1 |       4 |                |
|           |                      |
| |               +----------------+----------------+------+---------+----------------+
|           |                      |
| +AllNodesScan   | p              |             10 |    4 |       5 |            120 |
9160/0 |   108.923 | Fused in Pipeline 0 |
+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+

Total database accesses: 12, total allocated memory: 184
*/
```

For more information and examples, see Basic query tuning.

Notifications

After executing a query, the server can return notifications alongside the query result. Notifications contain recommendations for performance improvements, warnings about the usage of deprecated features, and other hints about sub-optimal usage of Neo4j.

For driver version >= 5.25 and server version >= 5.23, two forms of notifications are available (*Neo4j status codes* and *GQL status codes*). For earlier versions, only *Neo4j status codes* are available.
GQL status codes are planned to supersede Neo4j status codes.

Example 1. An unbounded shortest path raises a performance notification

Filter notifications

By default, the server analyses each query for all categories and severity of notifications. Starting from

version 5.7, you can use the parameters `NotificationsMinSeverity` and/or `NotificationsDisabledCategories`/`NotificationsDisabledClassifications` to restrict the severity and/or category/classification of notifications that you are interested into. There is a slight performance gain in restricting the amount of notifications the server is allowed to raise.

The severity filter applies to both Neo4j and GQL notifications. Category and classification filters exist separately only due to the discrepancy in lexicon between GQL and Neo4j; both filters affect either form of notification though, so you should use only one of them. You can use any of those parameters either when creating a `Driver` instance, or when creating a session.

You can disable notifications altogether by setting the minimum severity to `"OFF"`.

Allow only `Warning` notifications, but not of `Hint` or `Generic` category

```
// import (
//     "github.com/neo4j/neo4j-go-driver/v5/neo4j/notifications"
//     "github.com/neo4j/neo4j-go-driver/v5/neo4j/config"
// )

// At driver level
driverNot, _ := neo4j.NewDriverWithContext(
    dbUri,
    neo4j.BasicAuth(dbUser, dbPassword, ""),
    func (conf *config.Config) {
        conf.NotificationsMinSeverity = notifications.WarningLevel  // or "OFF" to disable entirely
        conf.NotificationsDisabledClassifications = notifications.DisableClassifications(notifications
.Hint, notifications.Generic)  // filters categories as well
    })


// At session level
sessionNot := driver.NewSession(ctx, neo4j.SessionConfig{
    NotificationsMinSeverity: notifications.WarningLevel,  // or "OFF" to disable entirely
    NotificationsDisabledClassifications: notifications.DisableClassifications(notifications.Hint,
notifications.Generic),  // filters categories as well
    DatabaseName: "neo4j",  // always provide the database name
})
```

Coordinate parallel transactions

When working with a Neo4j cluster, causal consistency is enforced by default in most cases, which guarantees that a query is able to read changes made by previous queries. The same does not happen by default for multiple transactions running in parallel though. In that case, you can use *bookmarks* to have one transaction wait for the result of another to be propagated across the cluster before running its own work. This is not a requirement, and **you should only use bookmarks if you need casual consistency across different transactions**, as waiting for bookmarks can have a negative performance impact.

A *bookmark* is a token that represents some state of the database. By passing one or multiple bookmarks along with a query, the server will make sure that the query does not get executed before the represented state(s) have been established.

Bookmarks with `ExecuteQuery()`

When querying the database with `ExecuteQuery()`, the driver manages bookmarks for you. In this case, you have the guarantee that subsequent queries can read previous changes without taking further action.

```
neo4j.ExecuteQuery(ctx, driver, "<QUERY 1>", nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))

// subsequent ExecuteQuery calls will be causally chained

neo4j.ExecuteQuery(ctx, driver, "<QUERY 2>", nil,  // can read result of <QUERY 1>
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
neo4j.ExecuteQuery(ctx, driver, "<QUERY 3>", nil,  // can read result of <QUERY 2>
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

To disable bookmark management and causal consistency, use the configuration callback `neo4j.ExecuteQueryWithoutBookmarkManager()` in `ExecuteQuery()` calls.

```
neo4j.ExecuteQuery(
    ctx, driver, "<QUERY>", nil, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"),
    neo4j.ExecuteQueryWithoutBookmarkManager())
```

Bookmarks within a single session

Bookmark management happens automatically for queries run within a single session, so that you can trust that queries inside one session are causally chained.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
session.ExecuteWrite(ctx,
    func(tx neo4j.ManagedTransaction) (any, error) {
        return tx.Run(ctx, "<QUERY 1>", nil)
})
session.ExecuteWrite(ctx,
    func(tx neo4j.ManagedTransaction) (any, error) {
        return tx.Run(ctx, "<QUERY 2>", nil)  // can read QUERY 1
})
session.ExecuteWrite(ctx,
    func(tx neo4j.ManagedTransaction) (any, error) {
        return tx.Run(ctx, "<QUERY 3>", nil)  // can read QUERY 1 and 2
})
```

Bookmarks across multiple sessions

If your application uses multiple sessions, you may need to ensure that one session has completed all its transactions before another session is allowed to run its queries.

In the example below, `sessionA` and `sessionB` are allowed to run concurrently, while `sessionC` waits until their results have been propagated. This guarantees the `Person` nodes `sessionC` wants to act on actually exist.

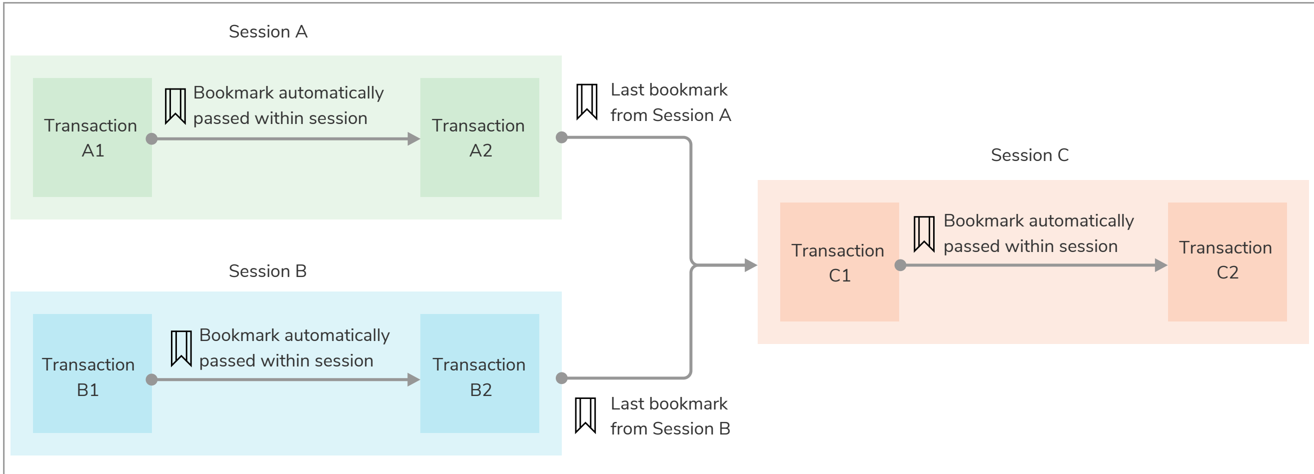
Coordinate multiple sessions using bookmarks

```
package main

import (
    "fmt"
    "context"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)
    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }

    // Bookmarks holder
    var savedBookmarks neo4j.Bookmarks

    // All function calls below may return errors,
    // we don't catch them here for simplicity.

    // Create the first person and employment relationship
    sessionA := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
    createPerson(ctx, sessionA, "Alice")
    employ(ctx, sessionA, "Alice", "Wayne Enterprises")
    savedBookmarks = neo4j.CombineBookmarks(savedBookmarks, sessionA.LastBookmarks())  ①
    sessionA.Close(ctx)

    // Create the second person and employment relationship
    sessionB := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
```

```
    createPerson(ctx, sessionB, "Bob")
    employ(ctx, sessionB, "Bob", "LexCorp")
    savedBookmarks = neo4j.CombineBookmarks(savedBookmarks, sessionB.LastBookmarks())  ①
    sessionB.Close(ctx)

    // Create a friendship between the two people created above
    sessionC := driver.NewSession(ctx, neo4j.SessionConfig{
        DatabaseName: "neo4j",
        Bookmarks: savedBookmarks,  ②
    })
    createFriendship(ctx, sessionC, "Alice", "Bob")
    printFriendships(ctx, sessionC)
}

// Create a Person node
func createPerson(ctx context.Context, session neo4j.SessionWithContext, name string) (any, error) {
    return session.ExecuteWrite(ctx,
        func(tx neo4j.ManagedTransaction) (any, error) {
            return tx.Run(ctx,
                "MERGE (:Person {name: $name})",
                map[string]any{"name": name})
        })
}

// Create an employment relationship to a pre-existing company node
// This relies on the person first having been created
func employ(ctx context.Context, session neo4j.SessionWithContext, personName string, companyName string)
(any, error) {
    return session.ExecuteWrite(ctx,
        func(tx neo4j.ManagedTransaction) (any, error) {
            return session.Run(ctx, `
                MATCH (person:Person {name: $person_name})
                MATCH (company:Company {name: $company_name})
                MERGE (person)-[:WORKS_FOR]->(company)
                `, map[string]any{
                    "personName": personName,
                    "companyName": companyName,
                })
        })
}

// Create a friendship between two people
func createFriendship(ctx context.Context, session neo4j.SessionWithContext, nameA string, nameB string)
(any, error) {
    return session.ExecuteWrite(ctx,
        func(tx neo4j.ManagedTransaction) (any, error) {
            return session.Run(ctx, `
                MATCH (a:Person {name: $nameA})
                MATCH (b:Person {name: $nameB})
                MERGE (a)-[:KNOWS]->(b)
                `, map[string]any{
                    "nameA": nameA,
                    "nameB": nameB,
                })
        })
}

// Retrieve and display all friendships
func printFriendships(ctx context.Context, session neo4j.SessionWithContext) (any, error) {
    return session.ExecuteRead(ctx,
        func(tx neo4j.ManagedTransaction) (any, error) {
            result, err := session.Run(ctx,
                "MATCH (a)-[:KNOWS]->(b) RETURN a.name, b.name",
                nil)
            if err != nil {
                return nil, err
            }
            records, _ := result.Collect(ctx)
            for _, record := range records {
                nameA, _ := record.Get("a.name")
                nameB, _ := record.Get("b.name")
                fmt.Println(nameA, "knows", nameB)
            }
            return nil, nil
        })
}
```

① Collect and combine bookmarks from different sessions using `SessionWithContext.LastBookmarks()` and `neo4j.CombineBookmarks()`, storing them in a `Bookmarks` object.

② Use them to initialize another session with the `Bookmarks` config parameter.



The use of bookmarks can negatively impact performance, since all queries are forced to wait for the latest changes to be propagated across the cluster. For simple use-cases, try to group queries within a single transaction, or within a single session.

Mix `ExecuteQuery()` and sessions

To ensure causal consistency among transactions executed partly with `ExecuteQuery()` and partly with sessions, you can use the parameter `BookmarkManager` upon session creation, setting it to `driver.ExecuteQueryBookmarkManager()`. Since that is the default bookmark manager for `ExecuteQuery()` calls, this will ensure that all work is executed under the same bookmark manager and thus causally consistent.

```
neo4j.ExecuteQuery(ctx, driver, "<QUERY 1>", nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))

session := driver.NewSession(ctx, neo4j.SessionConfig{
    DatabaseName: "neo4j",
    BookmarkManager: driver.ExecuteQueryBookmarkManager(),
})
// every query inside this session will be causally chained
// (i.e., can read what was written by <QUERY 1>)
session.ExecuteWrite(ctx,
    func(tx neo4j.ManagedTransaction) (any, error) {
        return tx.Run(ctx, "<QUERY 2>", nil)
})
session.Close(ctx)

// subsequent ExecuteQuery calls will be causally chained
// (i.e., can read what was written by <QUERY 2>)
neo4j.ExecuteQuery(ctx, driver, "<QUERY 3>", nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

Run concurrent transactions

You may leverage Goroutines and channels to run concurrent queries, or to delegate the processing of a query's result to multiple threads. The examples below also use the Go sync package to coordinate different routines. If you are not familiar with concurrency in Go, checkout The Go Programming Language → Go Concurrency Patterns: Pipelines and cancellation.

If you need causal consistency across different transactions, use bookmarks.

Concurrent processing of a query result set (using sessions)

The following example shows how you can stream a query result to a channel, and have its records concurrently processed by several consumers.

```
package main

import (
    "fmt"
    "context"
    "time"
    "sync"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)
    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }

    // Run a query and get results in a channel
    recordsC := queryToChannel(ctx, driver)  ①

    // Spawn some consumers that will process records
    // They communicate back on the log channel
    // WaitGroup allows to keep track of progress and close channel when all are done
    log := make(chan string)  ④
    wg := &sync.WaitGroup{}  ⑤
    for i := 1; i < 10; i++ {  // i starts from 1 because 0th receiver would process too fast
        wg.Add(1)
        go consumer(wg, recordsC, log, i)  ⑥
    }
    // When all consumers are done, close log channel
    go func() {
        wg.Wait()
        close(log)
    }()
    // Print log as it comes
    for v := range log {
        fmt.Println(v)
    }
}
```

```go
func queryToChannel(ctx context.Context, driver neo4j.DriverWithContext) chan *neo4j.Record {
    recordsC := make(chan *neo4j.Record, 10) ②
    session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
    defer session.Close(ctx)
    go session.ExecuteWrite(ctx,
        func(tx neo4j.ManagedTransaction) (any, error) {
            // Neo4j query to create and retrieve some nodes
            result, err := tx.Run(ctx, `
                UNWIND range(1,25) AS id
                MERGE (p:Person {id: id})
                RETURN p
                `, nil)
            if err != nil {
                panic(err)
            }
            // Stream results to channel as they come from the server
            for result.Next(ctx) { ③
                record := result.Record()
                recordsC <- record
            }
            close(recordsC)
            return nil, err
        })
    return recordsC
}

func consumer(wg *sync.WaitGroup, records <-chan *neo4j.Record, log chan string, n int) {
    defer wg.Done()  // will communicate that routine is done
    for record := range records {
        log <- fmt.Sprintf("Receiver %v processed %v", n, record)
        time.Sleep(time.Duration(n) * time.Second)  // proxy for a time-consuming processing
    }
}
```

① A Goroutine runs the query to the Neo4j server with a managed transaction. Notice that the driver session is created *inside* the routine, as sessions are not thread-safe.

② The channel `recordsC` is where the query result records get streamed to. The transaction function from `.ExecuteWrite()` writes to it, and the various `consumer`s read from it. It is buffered so that the driver does not retrieve records faster than what the consumers can handle.

③ Each result record coming from the server is sent over the `recordsC` channel. The streaming continues so long as there are records to be processed, after which the channel gets closed and the routine exits.

④ The channel `log` is where the consumers comunicate on.

⑤ A `sync.WaitGroup` is needed to know when all consumers are done, and thus the `log` channel can be closed.

⑥ A number of `consumer`s get started in separate Goroutines. Each consumer reads and processes records from the `recordsC` channel. Each consumer simulates a lengthy operation with a sleeping timer.

Concurrent run of multiple queries (using `ExecuteQuery()`)

The following example shows how you can run multiple queries concurrently.

```go
package main

import (
    "fmt"
    "context"
    "sync"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)
    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }

    log := make(chan string)  ①
    wg := &sync.WaitGroup{}  ②
    // Spawn 10 concurrent queries
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go runQuery(wg, ctx, driver, log)  ③
    }
    // Wait for all runner routines to be done before closing log
    go func() {
        wg.Wait()
        close(log)
    }()
    // Print log
    for msg := range log {
        fmt.Println(msg)
    }
}

// Run Neo4j query with random sleep time, returning the sleep time in ms
func runQuery(wg *sync.WaitGroup, ctx context.Context, driver neo4j.DriverWithContext, log chan string) {
    defer wg.Done()  // will communicate that routine is done
    result, err := neo4j.ExecuteQuery(ctx, driver, `
        WITH round(rand()*2000) AS waitTime
        CALL apoc.util.sleep(toInteger(waitTime)) RETURN waitTime AS time
        `, nil, neo4j.EagerResultTransformer,
        neo4j.ExecuteQueryWithDatabase("neo4j"))
    if err != nil {
        log <- fmt.Sprintf("ERROR: %v", err)
    } else {
        neo, _ := result.Records[0].Get("time")
        log <- fmt.Sprintf("Query returned %v", neo)
    }
}
```

① The `log` channel is where all query routine communicate to.

② A `sync.WaitGroup` is needed to know when all query routines are done, and thus the log channel can be closed.

③ Ten different queries are run, each in its own Go routine. They run independently and concurrently, reporting to the shared `log` channel.

Further query mechanisms

Implicit (or auto-commit) transactions

This is the most basic and limited form with which to run a Cypher query. The driver will not automatically retry implicit transactions, as it does instead for queries run with `ExecuteQuery()` and with managed transactions. Implicit transactions should only be used when the other driver query interfaces do not fit the purpose, or for quick prototyping.

You run an implicit transaction with the method `SessionWithContext.Run()`. It returns a `ResultWithContext` object that needs to be processed accordingly.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
result, err := session.Run(
    ctx,
    "CREATE (p:Person {name: $name}) RETURN p",
    map[string]any{
        "name": "Lucia",
    })
```

An implicit transaction gets committed *at the latest* when the session is destroyed, or before another transaction is executed within the same session. Other than that, there is no clear guarantee on when exactly an implicit transaction will be committed during the lifetime of a session. To ensure an implicit transaction is committed, you can call the `.Consume(ctx)` method on its result.

Since the driver cannot figure out whether the query in a `SessionWithContext.Run()` call requires a read or write session with the database, it defaults to write. If your implicit transaction contains read queries only, there is a performance gain in making the driver aware by setting the session config `AccessMode: neo4j.AccessModeRead` when creating the session.

Implicit transactions are the only ones that can be used for **`CALL { … } IN TRANSACTIONS`** queries.

Import CSV files

The most common use case for using `SessionWithContext.Run()` is for importing large CSV files into the database with the `LOAD CSV` Cypher clause, and preventing timeout errors due to the size of the transaction.

Import CSV data into a Neo4j database

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
result, err := session.Run(
    ctx, `
    LOAD CSV FROM 'https://data.neo4j.com/bands/artists.csv' AS line
    CALL {
        WITH line
        MERGE (:Artist {name: line[1], age: toInteger(line[2])})
    } IN TRANSACTIONS OF 2 ROWS
    `, nil)
summary, _ := result.Consume(ctx)
fmt.Println("Query updated the database?",
    summary.Counters().ContainsUpdates())
```

While `LOAD CSV` can be a convenience, there is nothing *wrong* in deferring the parsing of the CSV file to your Go application and avoiding `LOAD CSV`. In fact, moving the parsing logic to the application can give you more control over the importing process. For efficient bulk data insertion, see Performance → Batch data creation.

For more information, see Cypher → Clauses → Load CSV.

Transaction configuration

You can exert further control on implicit transactions by providing configuration callbacks after the third argument in `SessionWithContext.Run()` calls. The configuration callbacks allow to specify a query timeout and to attach metadata to the transaction. For more information, see Transactions — Transaction configuration.

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
people, err := session.Run(ctx,
    "MATCH (:Person) RETURN count(*) AS n",
    nil,
    neo4j.WithTxTimeout(5*time.Second),  // remember to import `time`
    neo4j.WithTxMetadata(map[string]any{"appName": "peopleTracker"}))
```

Dynamic values in property keys, relationship types, and labels

In general, you should not concatenate parameters directly into a query, but rather use query parameters. There can however be circumstances where your query structure prevents the usage of parameters in all its parts. In fact, although parameters can be used for literals and expressions as well as node and relationship ids, they cannot be used for the following constructs:

- property keys, so `MATCH (n) WHERE n.$param = 'something'` is invalid;
- relationship types, so `MATCH (n)-[:$param]→(m)` is invalid;
- labels, so `MATCH (n:$param)` is invalid.

For those queries, you are forced to use string concatenation. To protect against Cypher injections, you should enclose the dynamic values in backticks and escape them yourself. Notice that Cypher processes Unicode, so take care of the Unicode literal `\u0060` as well.

Manually escaping dynamic labels before concatenation

```
dangerousLabel := "Person\\u0060n"
// convert \u0060 to literal backtick and then escape backticks
// remember to import `strings`
escapedLabel := strings.ReplaceAll(dangerousLabel, "\\u0060", "`")
escapedLabel = strings.ReplaceAll(escapedLabel, "`", "``")

result, err := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:`" + escapedLabel + "` WHERE p.name = $name) RETURN p.name",
    map[string]any{
        "name": "Alice",
    },
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

Another workaround, which avoids string concatenation, is using the APOC procedure `apoc.merge.node`. It supports dynamic labels and property keys, but only for node merging.

Using `apoc.merge.node` to create a node with dynamic labels/property keys

```
propertyKey := "name"
label := "Person"

result, err := neo4j.ExecuteQuery(ctx, driver,
    "CALL apoc.merge.node($labels, $properties)",
    map[string]any{
        "labels": []string{label},
        "properties": map[string]any{propertyKey: "Alice"},
    },
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

If you are running Neo4j in Docker, APOC needs to be enabled when starting the container. See APOC → Installation → Docker.

Logging

The driver splits logging between driver events and Bolt events. To enable driver logging, use the `Config.Log` option when instantiating the driver:

```
// import "github.com/neo4j/neo4j-go-driver/v5/neo4j/config"

driver, err := neo4j.NewDriverWithContext(
    dbUri,
    neo4j.BasicAuth(dbUser, dbPassword, ""),
    func(conf *config.Config) {
        conf.Log = neo4j.ConsoleLogger(neo4j.DEBUG)
    })
```

Example of log output upon driver connection

```
2023-07-03 08:07:19.316   INFO  [pool 1] Created
2023-07-03 08:07:19.316   INFO  [router 1] Created {context: map[address:localhost:7687]}
2023-07-03 08:07:19.316   INFO  [driver 1] Created { target: localhost:7687 }
2023-07-03 08:07:19.316  DEBUG  [session 2] Created
2023-07-03 08:07:19.316   INFO  [router 1] Reading routing table from initial router: localhost:7687
2023-07-03 08:07:19.316  DEBUG  [pool 1] Trying to borrow connection from [localhost:7687]
2023-07-03 08:07:19.316   INFO  [pool 1] Connecting to localhost:7687
2023-07-03 08:07:19.320   INFO  [bolt5 bolt-58@localhost:7687] Connected
2023-07-03 08:07:19.320   INFO  [bolt5 bolt-58@localhost:7687] Retrieving routing table
2023-07-03 08:07:19.320  DEBUG  [pool 1] Returning connection to localhost:7687 {alive:true}
2023-07-03 08:07:19.320  DEBUG  [bolt5 bolt-58@localhost:7687] Resetting connection internal state
2023-07-03 08:07:19.320  DEBUG  [router 1] New routing table for 'neo4j', TTL 300
2023-07-03 08:07:19.320  DEBUG  [session 2] Resolved home database, uses db 'neo4j'
2023-07-03 08:07:19.320  DEBUG  [pool 1] Trying to borrow connection from [localhost:7687]
2023-07-03 08:07:19.321  DEBUG  [pool 1] Returning connection to localhost:7687 {alive:true}
2023-07-03 08:07:19.321  DEBUG  [bolt5 bolt-58@localhost:7687] Resetting connection internal state
2023-07-03 08:07:19.321  DEBUG  [router 1] Cleaning up
2023-07-03 08:07:19.321  DEBUG  [session 2] Closed
```

Bolt logging can be enabled either:

- per-query, with the configuration callback `neo4j.ExecuteQueryBoltLogger()`. This applies to individual queries run using `ExecuteQuery()`.
- per-session, with the configuration option `BoltLogger`. This applies to all queries within a session.

Enable logging for a query run with `ExecuteQuery`

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "RETURN 42 AS n", nil, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"),
    neo4j.ExecuteQueryWithBoltLogger(neo4j.ConsoleBoltLogger()))
```

Enable logging for a session

```
session := driver.NewSession(ctx, neo4j.SessionConfig{
    DatabaseName: "neo4j",
    BoltLogger: neo4j.ConsoleBoltLogger(),
})
defer session.Close(ctx)
session.Run(ctx, "RETURN 42 AS n", nil)
```

Example of Bolt logging output

```
2023-07-03 07:57:09.929   BOLT  [bolt-53@localhost:7687] C: BEGIN {"db":"neo4j"}
2023-07-03 07:57:09.930   BOLT  [bolt-53@localhost:7687] S: SUCCESS {}
2023-07-03 07:57:09.930   BOLT  [bolt-53@localhost:7687] C: RUN "RETURN 42 AS n" null null
2023-07-03 07:57:09.930   BOLT  [bolt-53@localhost:7687] C: PULL {"n":1000}
2023-07-03 07:57:09.936   BOLT  [bolt-53@localhost:7687] S: SUCCESS {"fields":["n"],"t_first":5}
2023-07-03 07:57:09.937   BOLT  [bolt-53@localhost:7687] S: RECORD [42]
2023-07-03 07:57:09.937   BOLT  [bolt-53@localhost:7687] S: SUCCESS {"t_first":1,"db":"neo4j"}
2023-07-03 07:57:09.937   BOLT  [bolt-53@localhost:7687] C: COMMIT
2023-07-03 07:57:09.938   BOLT  [bolt-53@localhost:7687] S: SUCCESS
{"bookmark":"FB:kcwQhRyDJPONRxudy+QyzPSuSAaQ"}
```

Performance recommendations

Always specify the target database

Specify the target database on all queries, either with the `ExecuteQueryWithDatabase()` configuration callback in `ExecuteQuery()` or with the `DatabaseName` configuration parameter when creating new sessions. If no database is provided, the driver has to send an extra request to the server to figure out what the default database is. The overhead is minimal for a single query, but becomes significant over hundreds of queries.

Good practices

```
result, err := neo4j.ExecuteQuery(ctx, driver, "<QUERY>", nil,
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{
    DatabaseName: "neo4j",
})
```

Bad practices

```
result, err := neo4j.ExecuteQuery(ctx, driver, "<QUERY>", nil,
    neo4j.EagerResultTransformer)
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{})
```

Be aware of the cost of transactions

When submitting queries through `ExecuteQuery()` or through `.ExecuteRead/Write()`, the server automatically wraps them into a transaction. This behavior ensures that the database always ends up in a consistent state, regardless of what happens during the execution of a transaction (power outages, software crashes, etc).

Creating a safe execution context around a number of queries yields an overhead that is not present if the driver just shoots queries at the server and hopes they will get through. The overhead is small, but can add up as the number of queries increases. For this reason, if your use case values throughput more than data integrity, you may extract further performance by running all queries within a single (auto-commit) transaction. You do this by creating a session and using `Session.Run()` to run as many queries as needed.

Privilege throughput over data integrity

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
for i := 0; i < 10000; i++ {
    session.Run(ctx, "<QUERY>", nil)
}
```

Privilege data integrity over throughput

```
for i := 0; i < 10000; i++ {
    neo4j.ExecuteQuery(ctx, driver, "<QUERY>", nil, neo4j.EagerResultTransformer)
    // or session.executeRead/Write() calls
}
```

Don’t fetch large result sets all at once

When submitting queries that may result in a lot of records, don’t retrieve them all at once. The Neo4j server can retrieve records in batches and stream the to the driver as they become available. Lazy-loading a result spreads out network traffic and memory usage.

For convenience, `.ExecuteQuery()` always retrieves all result records at once (it is what the `Eager` in `EagerResult` stands for). To lazy-load a result, you have to use `.ExecuteRead/Write()` (or other forms of manually-handled transactions) and **not** call `.Collect(ctx)` on the result; iterate on it instead.

Example 2. Comparison between eager and lazy loading

Eager loading	Lazy loading
• The server has to read all 250 records from the storage before it can send even the first one to the driver (i.e. it takes more time for the client to receive the first record). • Before any record is available to the application, the driver has to receive all 250 records. • The client has to hold in memory all 250 records.	• The server reads the first record and sends it to the driver. • The application can process records as soon as the first record is transferred. • Waiting time and resource consumption for the remaining records is deferred to when the application requests more records. • The server's fetch time can be used for client-side processing. • Resource consumption is bounded by the driver's fetch size.

Time and memory comparison between eager and lazy loading

```go
package main

import (
    "context"
    "time"
    "fmt"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

// Returns 250 records, each with properties
// - `output` (an expensive computation, to slow down retrieval)
// - `dummyData` (a list of 10000 ints, about 8 KB).
var slowQuery = `
UNWIND range(1, 250) AS s
RETURN reduce(s=s, x in range(1,1000000) | s + sin(toFloat(x))+cos(toFloat(x))) AS output,
range(1, 10000) AS dummyData
`
// Delay for each processed record
var sleepTime = "0.5s"

func main() {
    ctx := context.Background()
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, err := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(ctx)

    err = driver.VerifyConnectivity(ctx)
    if err != nil {
        panic(err)
    }

    log("LAZY LOADING (executeRead)")
    lazyLoading(ctx, driver)

    log("EAGER LOADING (executeQuery)")
    eagerLoading(ctx, driver)
}
```

```go
func lazyLoading(ctx context.Context, driver neo4j.DriverWithContext) {
    defer timer("lazyLoading")()

    sleepTimeParsed, err := time.ParseDuration(sleepTime)
    if err != nil {
        panic(err)
    }

    session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
    defer session.Close(ctx)
    session.ExecuteRead(ctx,
        func(tx neo4j.ManagedTransaction) (any, error) {
            log("Submit query")
            result, err := tx.Run(ctx, slowQuery, nil)
            if err != nil {
                return nil, err
            }
            for result.Next(ctx) != false {
                record := result.Record()
                output, _ := record.Get("output")
                log(fmt.Sprintf("Processing record %v", output))
                time.Sleep(sleepTimeParsed)  // proxy for some expensive operation
            }
            return nil, nil
        })
}

func eagerLoading(ctx context.Context, driver neo4j.DriverWithContext) {
    defer timer("eagerLoading")()

    log("Submit query")
    result, err := neo4j.ExecuteQuery(ctx, driver,
        slowQuery,
        nil,
        neo4j.EagerResultTransformer,
        neo4j.ExecuteQueryWithDatabase("neo4j"))
    if err != nil {
        panic(err)
    }

    sleepTimeParsed, err := time.ParseDuration(sleepTime)
    if err != nil {
        panic(err)
    }

    // Loop through results and do something with them
    for _, record := range result.Records {
        output, _ := record.Get("output")
        log(fmt.Sprintf("Processing record %v", output))
        time.Sleep(sleepTimeParsed)  // proxy for some expensive operation
    }
}

func log(msg string) {
    fmt.Println("[", time.Now().Unix(), "] ", msg)
}

func timer(name string) func() {
    start := time.Now()
    return func() {
        fmt.Printf("-- %s took %v --\n\n", name, time.Since(start))
    }
}
```

Output

```
[ 1718802595 ]  LAZY LOADING (executeRead)
[ 1718802595 ]  Submit query
[ 1718802595 ]  Processing record 0.5309371354666308  ①
[ 1718802595 ]  Processing record 1.5309371354662915
[ 1718802596 ]  Processing record 2.5309371354663197
...
[ 1718802720 ]  Processing record 249.53093713547042
-- lazyLoading took 2m5.467064085s --

[ 1718802720 ]  EAGER LOADING (executeQuery)
[ 1718802720 ]  Submit query
[ 1718802744 ]  Processing record 0.5309371354666308  ②
[ 1718802744 ]  Processing record 1.5309371354662915
[ 1718802745 ]  Processing record 2.5309371354663197
...
[ 1718802869 ]  Processing record 249.53093713547042
-- eagerLoading took 2m29.113482541s --  ③
```

① With lazy loading, the first record is quickly available.

② With eager loading, the first record is available ~25 seconds after the query has been submitted (i.e. after the server has retrieved all 250 records).

③ The total running time is lower with lazy loading, because while the client processes records the server can fetch the next ones. With lazy loading, the client could also stop requesting records after some condition is met (by calling `.Consume(ctx)` on the `Result`), saving time and resources.

The driver's fetch size affects the behavior of lazy loading. It instructs the server to stream an amount of records equal to the fetch size, and then wait until the client has caught up before retrieving and sending more.

The fetch size allows to bound memory consumption on the client side. It doesn't always bound memory consumption on the server side though: that depends on the query. For example, a query with `ORDER BY` requires the whole result set to be loaded into memory for sorting, before records can be streamed to the client.

The lower the fetch size, the more messages client and server have to exchange. Especially if the server's latency is high, a low fetch size may deteriorate performance.

Route read queries to cluster readers

In a cluster, **route read queries to** secondary nodes. You do this by:

- using the `ExecuteQueryWithReadersRouting()` configuration callback in `ExecuteQuery()` calls
- using `ExecuteRead()` instead of `ExecuteWrite()` (for managed transactions)
- setting `AccessMode: neo4j.AccessModeRead` when creating a new session (for explicit transactions).

Good practices

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person) RETURN p", nil, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"),
    neo4j.ExecuteQueryWithReadersRouting())
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
result, err := session.ExecuteRead(ctx,
    func(tx neo4j.ManagedTransaction) (any, error) {
        return tx.Run(ctx, "MATCH (p:Person) RETURN p", nil)
    })
```

Bad practices

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person) RETURN p", nil, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
    // defaults to routing = writers
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
result, err := session.ExecuteWrite(ctx,  // don't ask to write on a read-only operation
    func(tx neo4j.ManagedTransaction) (any, error) {
        return tx.Run(ctx, "MATCH (p:Person) RETURN p", nil)
    })
```

Create indexes

Create indexes for properties that you often filter against. For example, if you often look up `Person` nodes by the `name` property, it is beneficial to create an index on `Person.name`. You can create indexes with the `CREATE INDEX` Cypher clause, for both nodes and relationships.

```
// Create an index on Person.name
neo4j.ExecuteQuery(ctx, driver,
    "CREATE INDEX personName FOR (n:Person) ON (n.name)",
    nil, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

For more information, see Indexes for search performance.

Profile queries

Profile your queries to locate queries whose performance can be improved. You can profile queries by prepending them with `PROFILE`. The server output is available through the `.Profile()` method on the `ResultSummary` object.

```
result, _ := neo4j.ExecuteQuery(ctx, driver,
    "PROFILE MATCH (p {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    },
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
fmt.Println(result.Summary.Profile().Arguments()["string-representation"])
/*
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+
| Operator        | Details        | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache
Hits/Misses | Time (ms) | Pipeline            |
+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+
| +ProduceResults | p              |              1 |    1 |       3 |                |
|           |                      |
| |               +----------------+----------------+------+---------+----------------+
|           |                      |
| +Filter         | p.name = $name |              1 |    1 |       4 |                |
|           |                      |
| |               +----------------+----------------+------+---------+----------------+
|           |                      |
| +AllNodesScan   | p              |             10 |    4 |       5 |            120 |
9160/0 |   108.923 | Fused in Pipeline 0 |
+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+

Total database accesses: 12, total allocated memory: 184
*/
```

In case some queries are so slow that you are unable to even run them in reasonable times, you can prepend them with `EXPLAIN` instead of `PROFILE`. This will return the *plan* that the server would use to run the query, but without executing it. The server output is available through the `.Plan()` method on the `ResultSummary` object.

```
result, _ := neo4j.ExecuteQuery(ctx, driver,
    "EXPLAIN MATCH (p {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    },
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
fmt.Println(result.Summary.Plan().Arguments()["string-representation"])
/*
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+---------------------+
| Operator        | Details        | Estimated Rows | Pipeline            |
+-----------------+----------------+----------------+---------------------+
| +ProduceResults | p              |              1 |                     |
| |               +----------------+----------------+                     |
| +Filter         | p.name = $name |              1 |                     |
| |               +----------------+----------------+                     |
| +AllNodesScan   | p              |             10 | Fused in Pipeline 0 |
+-----------------+----------------+----------------+---------------------+

Total database accesses: ?
*/
```

Specify node labels

Specify node labels in all queries. This allows the query planner to work much more efficiently, and to leverage indexes where available. To learn how to combine labels, see Cypher → Label expressions.

Good practices

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person|Animal {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
result, err := session.Run(ctx,
    "MATCH (p:Person|Animal {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    })
```

Bad practices

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
result, err := session.Run(ctx,
    "MATCH (p {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    })
```

Batch data creation

Batch queries when creating a lot of records using the `UNWIND` Cypher clauses.

Good practice

```
numbers := make([]int, 10000)
for i := range numbers { numbers[i] = i }
neo4j.ExecuteQuery(ctx, driver, `
    WITH $numbers AS batch
    UNWIND batch AS value
    MERGE (n:Number)
    SET n.value = value
    `, map[string]any{
        "numbers": numbers,
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

Bad practice

```
for i := 0; i < 10000; i++ {
    neo4j.ExecuteQuery(ctx, driver,
        "MERGE (:Number {value: $value})",
        map[string]any{
            "value": i,
        }, neo4j.EagerResultTransformer,
        neo4j.ExecuteQueryWithDatabase("neo4j"))
}
```

The most efficient way of performing a *first import* of large amounts of data into a new database is the `neo4j-admin database import` command.

Use query parameters

Always use query parameters instead of hardcoding or concatenating values into queries. Besides protecting from Cypher injections, this allows to leverage the database query cache.

Good practices

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person {name: $name}) RETURN p",
    map[string]any{
        "name": "Alice",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
session.Run(ctx, "MATCH (p:Person {name: $name}) RETURN p", map[string]any{
    "name": "Alice",
})
```

Bad practices

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    "MATCH (p:Person {name: 'Alice'}) RETURN p",
    // or "MATCH (p:Person {name: '" + name + "'}) RETURN p"
    nil, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
```

```
session := driver.NewSession(ctx, neo4j.SessionConfig{DatabaseName: "neo4j"})
defer session.Close(ctx)
session.Run(ctx, "MATCH (p:Person {name: $name}) RETURN p", nil)
           // or "MATCH (p:Person {name: '" + name + "'}) RETURN p"
```

Concurrency

Use concurrency patterns. This is likely to be more impactful on performance if you parallelize complex and time-consuming queries in your application, but not so much if you run many simple ones.

Use MERGE for creation only when needed

The Cypher clause MERGE is convenient for data creation, as it allows to avoid duplicate data when an exact clone of the given pattern exists. However, it requires the database to run two queries: it first needs to MATCH the pattern, and only then can it CREATE it (if needed).

If you know already that the data you are inserting is new, avoid using MERGE and use CREATE directly instead — this practically halves the number of database queries.

Filter notifications

Filter the category and/or severity of notifications the server should raise.

=Reference=

Advanced connection information

Connection URI

The driver supports connection to URIs of the form

```
<SCHEME>://<HOST>[:<PORT>[?policy=<POLICY-NAME>]]
```

- `<SCHEME>` is one among `neo4j`, `neo4j+s`, `neo4j+ssc`, `bolt`, `bolt+s`, `bolt+ssc`.
- `<HOST>` is the host name where the Neo4j server is located.
- `<PORT>` is optional, and denotes the port the Bolt protocol is available at.
- `<POLICY-NAME>` is an optional *server policy* name. Server policies need to be set up prior to usage.

The driver does not support connection to a nested path, such as `example.com/neo4j/`. The server must be reachable from the domain root.

Connection protocols and security

Communication between the driver and the server is mediated by Bolt. The scheme of the server URI determines whether the connection is encrypted and, if so, what type of certificates are accepted.

URL scheme	Encryption	Comment
neo4j	✗	Default for local setups
neo4j+s	✔ (only CA-signed certificates)	Default for Aura
neo4j+ssc	✔ (CA- and self-signed certificates)	

The driver receives a *routing table* from the server upon successful connection, regardless of whether the instance is a proper cluster environment or a single-machine environment. The driver's routing behavior works in tandem with Neo4j's clustering by directing read/write transactions to appropriate cluster members. If you want to target a *specific* machine, use the `bolt`, `bolt+s`, or `bolt+ssc` URI schemes instead.

The connection scheme to use is not your choice, but is rather determined by the server requirements. You must know the right server scheme upfront, as no metadata is exposed prior to connection. If you are unsure, ask the database administrator.

Authentication methods

Basic authentication

The basic authentication scheme relies on traditional username and password. These can either be the credentials for your local installation, or the ones provided with an Aura instance.

```
driver, err := neo4j.NewDriverWithContext(
    dbUri,
    neo4j.BasicAuth(dbUser, dbPassword, ""))
```

The basic authentication scheme can also be used to authenticate against an LDAP server (Enterprise Edition only).

Kerberos authentication

The Kerberos authentication scheme requires a base64-encoded ticket. It can only be used if the server has the Kerberos Add-on installed.

```
1 driver, err := neo4j.NewDriverWithContext(dbUri, neo4j.KerberosAuth(ticket))
```

Bearer authentication

The bearer authentication scheme requires a base64-encoded token provided by an Identity Provider through Neo4j's Single Sign-On feature.

```
1 driver, err := neo4j.NewDriverWithContext(dbUri, neo4j.BearerAuth(token))
```

The bearer authentication scheme requires configuring Single Sign-On on the server. Once configured, clients can discover Neo4j's configuration through the Discovery API.

Custom authentication

Use the function `CustomAuth` to log into a server having a custom authentication scheme.

No authentication

Use the function `NoAuth` to access a server where authentication is disabled.

```
1 driver, err := neo4j.NewDriverWithContext(dbUri, neo4j.NoAuth())
```

Custom address resolver

When creating a `DriverWithContext` object, you can specify a *resolver* function to resolve the connection address the driver is initialized with. Note that addresses that the driver receives in routing tables are not resolved with the custom resolver. Your resolver function is called with a `ServerAddress` object and should return a list of `ServerAddress` objects.

Connection to `example.com` on port `9999` is resolved to `localhost` on port `7687`

```
// import "github.com/neo4j/neo4j-go-driver/v5/neo4j/config"

driver, err := neo4j.NewDriverWithContext(
    "neo4j://example.com:9999", neo4j.BasicAuth(dbUser, dbPassword, ""),
    func(conf *config.Config) {
        conf.AddressResolver = func(address config.ServerAddress) []config.ServerAddress {
            return []config.ServerAddress{
                neo4j.NewServerAddress("localhost", "7687"),
            }
        }
})
defer driver.Close(ctx)
```

Further connection parameters

You can find all `DriverWithContext` configuration parameters in the API documentation → config package.

Data types and mapping to Cypher types

The tables in this section show the mapping between Cypher data types and Go types.

When accessing a record's content, all its properties are of type any. This means that you have to cast them to the relevant Go type if you want to use methods/features defined on such types. For example, if the `name` property coming from the database is a string, `record.AsMap()["name"][1]` would result in an *invalid operation* error at compilation time. For it to work, cast the value to string *before* using it as a string: `name := record.AsMap()["name"].(string)` and then `name[1]`.

Core types

Cypher Type	Go Type
NULL	`nil`
`LIST`	`[]any`
`MAP`	`map[string]any`
`BOOLEAN`	`bool`
`INTEGER`	`int64`
`FLOAT`	`float64`
`STRING`	`string`
`ByteArray`	`[]byte`

Temporal types

The driver provides a set of temporal data types compliant with ISO-8601 and Cypher. Sub-second values are measured to nanosecond precision.

The driver's types rely on Go's `time` types. All temporal types, except `neo4j.Duration`, are in fact `time.Date` objects under the hood. This means that:

- if you want to *query* the database with a temporal type, instantiate a `time.Date` object and use it as query parameter (i.e. you don't need to care about driver's types)
- if you *retrieve* a temporal object that you had previously inserted starting from a `time.Date` object, you will get back a `time.Date` object (i.e. you don't need to care about driver's types)
- if you receive a temporal object using one of Cypher temporal functions, you will get back the corresponding *driver type* as displayed in the table below. You may then use `.Time()` on them to convert them into Go `time.Date` objects.

Cypher Type	Go Type
`DATE`	`neo4j.Date`
`ZONED TIME`	`neo4j.OffsetTime`

Cypher Type	Go Type
LOCAL TIME	neo4j.LocalTime
ZONED DATETIME	neo4j.Time
LOCAL DATETIME	neo4j.LocalDateTime
DURATION	neo4j.Duration

Using temporal types in queries

```go
package main

import (
    "fmt"
    "context"
    "time"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
    "reflect"
)

func main() {
    ctx := context.Background()

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, _ := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    driver.VerifyConnectivity(ctx)

    // Define a date, with timezone
    location, _ := time.LoadLocation("Europe/Stockholm")
    friendsSince := time.Date(2006, time.December, 16, 13, 59, 59, 999999999, location)

    result, err := neo4j.ExecuteQuery(ctx, driver, `
        MERGE (a:Person {name: $name})
        MERGE (b:Person {name: $friend})
        MERGE (a)-[friendship:KNOWS {since: $friendsSince}]->(b)
        RETURN friendship.since AS date
        `, map[string]any{
            "name": "Alice",
            "friend": "Bob",
            "friendsSince": friendsSince,
        }, neo4j.EagerResultTransformer,
        neo4j.ExecuteQueryWithDatabase("neo4j"))
    if err != nil {
        panic(err)
    }
    date, _ := result.Records[0].Get("date")
    fmt.Println(reflect.TypeOf(date))  // time.Time
    fmt.Println(date)  // 2006-12-16 13:59:59.999999999 +0200 EET
}
```

Using driver's temporal types

```
package main

import (
    "fmt"
    "context"
    "time"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
    "reflect"
)

func main() {
    ctx := context.Background()

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, _ := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    driver.VerifyConnectivity(ctx)

    // Query and return a neo4j.Time object
    result, err := neo4j.ExecuteQuery(ctx, driver, `
        MERGE (a:Person {name: $name})
        MERGE (b:Person {name: $friend})
        MERGE (a)-[friendship:KNOWS {since: time()}]->(b)
        RETURN friendship.since AS time
        `, map[string]any{
            "name": "Alice",
            "friend": "Sofia",
        }, neo4j.EagerResultTransformer,
        neo4j.ExecuteQueryWithDatabase("neo4j"))
    if err != nil {
        panic(err)
    }
    time, _ := result.Records[0].Get("time")
    fmt.Println(reflect.TypeOf(time))  // time.Time
    castDate, _ := time.(neo4j.Time)  // cast from `any` to `neo4j.Time`
    fmt.Println(castDate.Time())  // -0001-11-30 12:18:08.973 +0000 Offset
}
```

Duration

Represents the difference between two points in time.

```
duration := neo4j.Duration{
    Months: 1,
    Days: 2,
    Seconds: 3,
    Nanos: 4,
}
fmt.Println(duration)  // 'P1Y2DT3.000000004S'
```

For full documentation, see API documentation → Duration.

Spatial types

Cypher supports spatial values (points), and Neo4j can store these point values as properties on nodes and relationships.

The object attribute `SpatialRefId` (short for *Spatial Reference Identifier*) is a number identifying the coordinate system the spatial type is to be interpreted in. You can think of it as a unique identifier for each

spatial type.

Cypher Type	Go Type	SpatialRefId
POINT (2D Cartesian)	neo4j.Point2D	7203
POINT (2D WGS-84)	neo4j.Point2D	4326
POINT (3D Cartesian)	neo4j.Point3D	9157
POINT (3D WGS-84)	neo4j.Point3D	4979

Spatial types are implemented in the dbtype package, so that the actual types are dbtype.Point2D/3D. However, they are also imported in the main neo4j package, so that they can also be used as neo4j.Point2D/3D.

Point2D

The type Point2D can be used to represent either a 2D Cartesian point or a 2D World Geodetic System (WGS84) point, depending on the value of SpatialRefId.

```
// A 2D Cartesian Point
cartesian2d := neo4j.Point2D{
    X:            1.23,
    Y:            4.56,
    SpatialRefId: 7203,
}
fmt.Println(cartesian2d)
// Point{srId=7203, x=1.230000, y=4.560000}

// A 2D WGS84 Point
wgs842d := neo4j.Point2D{
    X:            1.23,
    Y:            4.56,
    SpatialRefId: 9157,
}
fmt.Println(wgs842d)
// Point{srId=9157, x=1.230000, y=4.560000}
```

Point3D

The type Point3D can be used to represent either a 3D Cartesian point or a 3D World Geodetic System (WGS84) point, depending on the value of SpatialRefId.

```
// A 3D Cartesian Point
cartesian3d := neo4j.Point3D{
    X:            1.23,
    Y:            4.56,
    Z:            7.89,
    SpatialRefId: 9157,
}
fmt.Println(cartesian3d)
// Point{srId=9157, x=1.230000, y=4.560000, z=7.890000}

// A 3D WGS84 Point
wgs843d := neo4j.Point3D{
    X:            1.23,
    Y:            4.56,
    Z:            7.89,
    SpatialRefId: 4979,
}
fmt.Println(wgs843d)
// Point{srId=4979, x=1.230000, y=4.560000, z=7.890000}
```

Graph types

Graph types are only returned as query results and may not be used as parameters.

Cypher Type	Python Type
NODE	dbtype.Node
RELATIONSHIP	dbtype.Relationship
PATH	dbtype.Path

Node

Represents a node in a graph.
The property `ElementId` contains the database internal identifier for the entity. This should be used with care, as no guarantees are given about the mapping between id values and elements outside the scope of a single transaction. In other words, using an `ElementId` to `MATCH` an element across different transactions is risky.

```
result, err := neo4j.ExecuteQuery(ctx, driver, `
    MERGE (p:Person {name: $name}) RETURN p AS person, p.name as name
    `, map[string]any{
        "name": "Alice",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
if err != nil {
    panic(err)
}
node, _ := result.Records[0].AsMap()["person"].(neo4j.Node)
fmt.Println("Node ID:", node.ElementId)
fmt.Println("Node labels:", node.Labels)
fmt.Println("Node properties:", node.Props)

// Node ID: 4:2691aa68-87cc-467d-9d09-431df9f5c456:0
// Node labels: [Person]
// Node properties: map[name:Alice]
```

For full documentation, see API documentation → Node.

Relationship

Represents a relationship in a graph.
The property `ElementId` contains the database internal identifier for the entity. This should be used with care, as no guarantees are given about the mapping between id values and elements outside the scope of a single transaction.

```
result, err := neo4j.ExecuteQuery(ctx, driver, `
    MERGE (p:Person {name: $name})
    MERGE (p)-[r:KNOWS {status: $status, since: date()}]->(friend:Person {name: $friendName})
    RETURN r AS friendship
    `, map[string]any{
        "name": "Alice",
        "status": "BFF",
        "friendName": "Bob",
    }, neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("neo4j"))
if err != nil {
    panic(err)
}
relationship, _ := result.Records[0].AsMap()["friendship"].(neo4j.Relationship)
fmt.Println("Relationship ID:", relationship.ElementId)
fmt.Println("Relationship type:", relationship.Type)
fmt.Println("Relationship properties:", relationship.Props)
fmt.Println("Relationship start elID:", relationship.StartElementId)
fmt.Println("Relationship end elID:", relationship.EndElementId)

// Relationship ID: 5:2691aa68-87cc-467d-9d09-431df9f5c456:0
// Relationship type: KNOWS
// Relationship properties: map[since:{0 63824025600 <nil>} status:BFF]
// Relationship start elID: 4:2691aa68-87cc-467d-9d09-431df9f5c456:0
// Relationship end elID: 4:2691aa68-87cc-467d-9d09-431df9f5c456:1
```

For full documentation, see API documentation → Relationship.

Path

Represents a path in a graph.

Example of path creation, retrieval, and processing

```
package main

import (
    "fmt"
    "context"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    ctx := context.Background()

    // Connection to database
    dbUri := "<URI for Neo4j database>"
    dbUser := "<Username>"
    dbPassword := "<Password>"
    driver, _ := neo4j.NewDriverWithContext(
        dbUri,
        neo4j.BasicAuth(dbUser, dbPassword, ""))
    driver.VerifyConnectivity(ctx)

    // Create some :Person nodes linked by :KNOWS relationships
    addFriend(ctx, driver, "Alice", "BFF", "Bob")
    addFriend(ctx, driver, "Bob", "Fiends", "Sofia")
    addFriend(ctx, driver, "Sofia", "Acquaintances", "Sofia")

    // Follow :KNOWS relationships outgoing from Alice three times, return as path
    result, err := neo4j.ExecuteQuery(ctx, driver, `
        MATCH path=(:Person {name: $name})-[:KNOWS*3]->(:Person)
        RETURN path AS friendshipChain
        `, map[string]any{
            "name": "Alice",
        }, neo4j.EagerResultTransformer,
        neo4j.ExecuteQueryWithDatabase("neo4j"))
    if err != nil {
        panic(err)
    }
    path := result.Records[0].AsMap()["friendshipChain"].(neo4j.Path)

    fmt.Println("-- Path breakdown --")
    for i := range path.Relationships {
        name := path.Nodes[i].Props["name"]
        status := path.Relationships[i].Props["status"]
        friendName := path.Nodes[i+1].Props["name"]
        fmt.Printf("%s is friends with %s (%s)\n", name, friendName, status)
    }
}

func addFriend(ctx context.Context, driver neo4j.DriverWithContext, name string, status string, friendName
string) {
    _, err := neo4j.ExecuteQuery(ctx, driver, `
        MERGE (p:Person {name: $name})
        MERGE (p)-[r:KNOWS {status: $status, since: date()}]->(friend:Person {name: $friendName})
        `, map[string]any{
            "name": name,
            "status": status,
            "friendName": friendName,
        }, neo4j.EagerResultTransformer,
        neo4j.ExecuteQueryWithDatabase("neo4j"))
    if err != nil {
        panic(err)
    }
}
```

For full documentation, see API documentation → Path.

Exceptions

For the most part, the driver simply forwards any error the server may raise. For a list of errors the server can return, see the Status code page.

Some server errors are marked as safe to retry without need to alter the original request. Examples of such errors are deadlocks, memory issues, or connectivity issues. When an error is raised, the function `neo4j.IsRetryable(error)` gives insights into whether a further attempt might be successful. This is particular useful when running queries in explicit transactions, to know if a failed query should be run again. Note that managed transactions already implement a retry mechanism, so you don't need to implement your own.

API documentation

=GraphAcademy courses=

Graph Data Modeling Fundamentals

Intermediate Cypher Queries

Building Neo4j Applications with Go

License

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms

Attribution

You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial

You may not use the material for commercial purposes.

ShareAlike

If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions

You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

See https://creativecommons.org/licenses/by-nc-sa/4.0/ for further details. The full license text is available at https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.