The Neo4j JavaScript Driver Manual v5.0

Table of Contents

Quickstart

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

Installation

Install the Neo4j Javascript driver with `npm`:

```
npm i neo4j-driver
```

More info on installing the driver →

Connect to the database

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

```
var neo4j = require('neo4j-driver');
(async () => {
  // URI examples: 'neo4j://localhost', 'neo4j+s://xxx.databases.neo4j.io'
  const URI = '<URI to Neo4j database>'
  const USER = '<Username>'
  const PASSWORD = '<Password>'
  let driver

  try {
    driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
    const serverInfo = await driver.getServerInfo()
    console.log('Connection established')
    console.log(serverInfo)
  } catch(err) {
    console.log(`Connection error\n${err}\nCause: ${err.cause}`)
  }
})();
```

More info on connecting to a database →

Query the database

Execute a Cypher statement with the method `Driver.executeQuery()`. Do not hardcode or concatenate parameters: use placeholders and specify the parameters as key-value pairs.

```
// Get the name of all 42 year-olds
const { records, summary, keys } = await driver.executeQuery(
  'MATCH (p:Person {age: $age}) RETURN p.name AS name',
  { age: 42 },
  { database: 'neo4j' }
)

// Summary information
console.log(
  `>> The query ${summary.query.text} ` +
  `returned ${records.length} records ` +
  `in ${summary.resultAvailableAfter} ms.`
)

// Loop through results and do something with them
console.log('>> Results')
for(record of records) {
  console.log(record.get('name'))
}
```

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

```
const neo4j = require('neo4j-driver');

(async () => {
  const URI = '<URI for Neo4j database>'
  const USER = '<Username>'
  const PASSWORD = '<Password>'
  let driver, session
  let employeeThreshold = 10

  try {
    driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
    await driver.verifyConnectivity()
  } catch(err) {
    console.log(`-- Connection error --\n${err}\n-- Cause --\n${err.cause}`)
    await driver.close()
    return
  }

  session = driver.session({ database: 'neo4j' })
  for(let i=0; i<100; i++) {
    const name = `Neo-${i.toString()}`
    const orgId = await session.executeWrite(async tx => {
      let result, orgInfo

      // Create new Person node with given name, if not already existing
      await tx.run(`
        MERGE (p:Person {name: $name})
        RETURN p.name AS name
        `, { name: name }
      )

      // Obtain most recent organization ID and number of people linked to it
      result = await tx.run(`
        MATCH (o:Organization)
        RETURN o.id AS id, COUNT{(p:Person)-[r:WORKS_FOR]->(o)} AS employeesN
        ORDER BY o.createdDate DESC
        LIMIT 1
      `)
      if(result.records.length > 0) {
        orgInfo = result.records[0]
```

```
      }

      if(orgInfo != undefined && orgInfo['employeesN'] == 0) {
        throw new Error('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 that
      if(orgInfo != undefined && orgInfo['employeesN'] < employeeThreshold) {
        result = await tx.run(`
          MATCH (o:Organization {id: $orgId})
          MATCH (p:Person {name: $name})
          MERGE (p)-[r:WORKS_FOR]->(o)
          RETURN $orgId AS id
          `, { orgId: orgInfo['id'], name: name }
        )

      // Otherwise, create a new Organization and link Person to it
      } else {
        result = await tx.run(`
          MATCH (p:Person {name: $name})
          CREATE (o:Organization {id: randomuuid(), createdDate: datetime()})
          MERGE (p)-[r:WORKS_FOR]->(o)
          RETURN o.id AS id
          `, { name: name }
        )
      }

      // Return the Organization ID to which the new Person ends up in
      return result.records[0].get('id')
    })
    console.log(`User ${name} added to organization ${orgId}`)
  }
  await session.close()
  await driver.close()
})()
```

More info on running transactions →

Close connections and sessions

Call the `.close()` method on the `Driver` instance when you are finished with it, to release any resources still held by it. The same applies to any open sessions.

```
const driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
let session = driver.session({ database: 'neo4j' })

// session/driver usage

session.close()
driver.close()
```

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.

Driver

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

=Basic workflow=

Installation

To create a Neo4j JavaScript application, you first need to install the JavaScript Driver and get a Neo4j database instance to connect to.

Install the driver

Use `npm` to install the Neo4j JavaScript Driver (requires `npm` and any LTS version of `node.js`):

```
npm i neo4j-driver
```

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.

There is also a lite version of the Neo4j JavaScript Driver, which includes the same features as the regular driver but without support for the Reactive API. You can install it with `npm i neo4j-driver-lite`.

Include the driver

Unless you are going to use the driver within a web browser, you use the JavaScript driver in a Node.js or TypeScript application. To include the driver in your application, use `require`:

```
const neo4j = require('neo4j-driver')
```

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 database by creating a Driver object and providing a URL and an authentication token.

```
(async () => {
  var neo4j = require('neo4j-driver')

  // URI examples: 'neo4j://localhost', 'neo4j+s://xxx.databases.neo4j.io'
  const URI = '<URI for Neo4j database>'
  const USER = '<Username>'
  const PASSWORD = '<Password>'
  let driver

  try {
    driver = neo4j.driver(URI,  neo4j.auth.basic(USER, PASSWORD))  ①
    const serverInfo = await driver.getServerInfo()  ②
    console.log('Connection established')
    console.log(serverInfo)
  } catch(err) {
    console.log(`Connection error\n${err}\nCause: ${err.cause}`)
    await driver.close()
    return
  }

  // Use the driver to run queries

  await driver.close()  ③
})();
```

① Creating a `Driver` 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 To verify immediately that the driver can connect to the database (valid credentials, compatible versions, etc), use the `.getServerInfo()` method after initializing the driver.

③ Always close `Driver` objects to free up all allocated resources, even upon unsuccessful connection or runtime errors in subsequent querying.

Both the creation of a `Driver` object and the connection verification can raise exceptions, so error catching should include both.

`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 `Driver` instance. If you want to alter a `Driver` 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 `dotenv-java` for that purpose.

```
(async () => {
  var neo4j = require('neo4j-driver')
  require('dotenv').config({
    path: 'Neo4j-a0a2fa1d-Created-2023-11-06.txt',
    debug: true  // to raise file/parsing errors
  })

  const URI = process.env.NEO4J_URI
  const USER = process.env.NEO4J_USERNAME
  const PASSWORD = process.env.NEO4J_PASSWORD
  let driver

  try {
    driver = neo4j.driver(URI,  neo4j.auth.basic(USER, PASSWORD))
    const serverInfo = await driver.getServerInfo()
    console.log('Connection established')
    console.log(serverInfo)
  } catch(err) {
    console.log(`Connection error\n${err}\nCause: ${err.cause}`)
    await driver.close()
    return
  }

  // Use the driver to run queries

  await driver.close()
})();
```

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.

Close connections

Always close `Driver` objects via `Driver.close()` to free up all allocated resources, even upon unsuccessful connection or runtime errors.

Further connection parameters

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

Query the database

Once you have connected to the database, you can run Cypher queries through the method `Driver.executeQuery()`.

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

Due to the usage of `async/await`, the examples in this page need to be wrapped in an `async` function. See a full example if you are unsure how to do it.

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`

```
let { records, summary } = await driver.executeQuery(
  'CREATE (p:Person {name: $name})',  ①
   { name: 'Alice' },  ②
   { database: 'neo4j' }  ③
)
console.log(
  `Created ${summary.counters.updates().nodesCreated} nodes ` +
  `in ${summary.resultAvailableAfter} ms.`
)
```

① The Cypher query.

② An object of query parameters.

③ Which database the query should be run against.

Read from the database

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

Retrieve all `Person` nodes

```
let { records, summary } = await driver.executeQuery(
  'MATCH (p:Person) RETURN p.name AS name',
  {},
  { database: 'neo4j' }
)

// Loop through users and do process them
for(let record of records) {  ①
  console.log(`Person with name: ${record.get('name')}`)
  console.log(`Available properties for this node are: ${record.keys}\n`)
}

// Summary information
console.log(  ②
  `The query \`${summary.query.text}\` ` +
  `returned ${records.length} nodes.\n`
)
```

① `records` contains the actual result as a list of `Record` objects.

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

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

```
let { _, summary } = await driver.executeQuery(`
  MATCH (p:Person {name: $name})
  SET p.age = $age
  `, { name: 'Alice', age: 42 },
  { database: 'neo4j' }
)
console.log('Query counters:')
console.log(summary.counters.updates())
```

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`

```
let { records, summary } = await driver.executeQuery(`
  MATCH (alice:Person {name: $name})  ①
  MATCH (bob:Person {name: $friendName})  ②
  CREATE (alice)-[:KNOWS]->(bob)  ③
  `, { name: 'Alice', friendName: 'Bob' },
  { database: 'neo4j' }
)
console.log('Query counters:')
console.log(summary.counters.updates())
```

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

```
let { _, summary } = await driver.executeQuery(`
  MATCH (p:Person WHERE p.name = $name)
  DETACH DELETE p
  `, { name: 'Alice' },
  { database: 'neo4j' }
)
console.log('Query counters:')
console.log(summary.counters.updates())
```

Query parameters

Do not hardcode or concatenate parameters directly into queries. Instead, always use placeholders and specify the Cypher parameters as keyword arguments or in a dictionary, 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**: protecting against Cypher injection.

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.

Error handling

To avoid an error in one query crashing your application, you can wrap queries into `try/catch` blocks. We avoid proper error handling throughout this manual to make examples lighter to parse, and because appropriate error handling depends on the application. Here below an example with a `try/catch` block.

```
try {
  let result = await driver.executeQuery('MATCH (p:Person) RETURN p')
} catch(err) {
  console.log(`Error in query\n${err}`)
}
```

The driver automatically retries to run a failed query, if the failure is deemed to be transient (for example due to temporary server unavailability). An exception will be raised if the operation keeps failing after a number of attempts.

Query configuration

You can supply a `QueryConfig` object as third (optional) parameter to alter the default behavior of `.executeQuery()`.

Database selection

It is recommended to **always specify the database explicitly** with the `database` parameter, 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.

```
await driver.executeQuery(
  'MATCH (p:Person) RETURN p.name',
  {},
  {
    database: 'neo4j'
  }
)
```

Specifying the database through the configuration parameter 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 configuration `routing: 'READ'` to route a query to the read nodes.

```
await driver.executeQuery(
  'MATCH (p:Person) RETURN p.name',
  {},
  {
    routing: 'READ',  // short for neo4j.routing.READ
    database: 'neo4j'
  }
)
```

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 parameter `impersonatedUser`, specifying the name of the user to impersonate. For this to work, the user under which the `Driver` was created needs to have the appropriate permissions. Impersonating a user is cheaper than creating a new `Driver` object.

```
await driver.executeQuery(
  'MATCH (p:Person) RETURN p.name',
  {},
  {
    impersonatedUser: 'somebodyElse',
    database: 'neo4j'
  }
)
```

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

```
const neo4j = require('neo4j-driver');

(async () => {
  const URI = '<URI for Neo4j database>'
  const USER = '<Username>'
  const PASSWORD = '<Password>'
  let driver, result

  let people = [{name: 'Alice', age: 42, friends: ['Bob', 'Peter', 'Anna']},
                {name: 'Bob', age: 19},
                {name: 'Peter', age: 50},
                {name: 'Anna', age: 30}]

  // Connect to database
  try {
    driver = neo4j.driver(URI,  neo4j.auth.basic(USER, PASSWORD))
    await driver.verifyConnectivity()
  } catch(err) {
    console.log(`Connection error\n${err}\nCause: ${err.cause}`)
    await driver.close()
    return
  }

  // Create some nodes
  for(let person of people) {
    await driver.executeQuery(
      'MERGE (p:Person {name: $person.name, age: $person.age})',
      { person: person },
      { database: 'neo4j' }
    )
  }

  // Create some relationships
  for(let person of people) {
    if(person.friends != undefined) {
      await driver.executeQuery(`
        MATCH (p:Person {name: $person.name})
        UNWIND $person.friends AS friendName
        MATCH (friend:Person {name: friendName})
        MERGE (p)-[:KNOWS]->(friend)
        `, { person: person },
        { database: 'neo4j' }
      )
    }
  }

  // Retrieve Alice's friends who are under 40
  result = await driver.executeQuery(`
    MATCH (p:Person {name: $name})-[:KNOWS]-(friend:Person)
    WHERE friend.age < $age
    RETURN friend
    `, { name: 'Alice', age: 40 },
    { database: 'neo4j' }
  )

  // Loop through results and do something with them
  for(let person of result.records) {
    // `person.friend` is an object of type `Node`
    console.log(person.get('friend'))
  }

  // Summary information
  console.log(
    `The query \`${result.summary.query.text}\` ` +
    `returned ${result.records.length} records ` +
    `in ${result.summary.resultAvailableAfter} ms.`
  )

  await driver.close()
})();
```

=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 `execute_query()` 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 `Driver.session()`. It takes a single (optional) object parameter, with the property `database` allowing to specify the target database. For further parameters, see Session configuration.

```
session = driver.session({ database: 'neo4j' })
```

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 `Driver` object across threads, but make sure each thread 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 `Session.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 of actually carrying out the queries and processing the result.

Retrieve people whose name starts with `Al`.

```
let session = driver.session({ database: 'neo4j' })  ①
try {
  let result = await session.executeRead(async tx => { ②
    return await tx.run(`  ③
      MATCH (p:Person) WHERE p.name STARTS WITH $filter
      RETURN p.name AS name ORDER BY name
      `, {filter: 'Al'}
    )
  })
  for(let record in result.records) {  ④
    console.log(record.get('name'))
  }
  console.log(
    `The query \`${result.summary.query.text}\`` +
    `returned ${result.records.length} nodes.\n`
  )
} finally {
  session.close()
}
```

① Create a session. A single session can be the container for multiple queries. Remember to close it when done.

② The `.executeRead()` (or `.executeWrite()`) method is the entry point into a transaction.

③ Use the method `Result` object.

④ Process the result records and query summary.

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; at minimum, cast it to list. Within a transaction function, a `return` statement results in the transaction being committed, while the transaction is automatically rolled back if an exception is raised.

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

```
const neo4j = require('neo4j-driver');

(async () => {
  const URI = '<URI for Neo4j database>'
  const USER = '<Username>'
  const PASSWORD = '<Password>'
  let driver, session
  let employeeThreshold = 10

  try {
    driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
    await driver.verifyConnectivity()
  } catch(err) {
    console.log(`-- Connection error --\n${err}\n-- Cause --\n${err.cause}`)
    await driver.close()
    return
  }

  session = driver.session({ database: 'neo4j' })
  for(let i=0; i<100; i++) {
    const name = `Neo-${i.toString()}`
    const orgId = await session.executeWrite(async tx => {
```

```
      let result, orgInfo

      // Create new Person node with given name, if not already existing
      await tx.run(`
        MERGE (p:Person {name: $name})
        RETURN p.name AS name
        `, { name: name }
      )

      // Obtain most recent organization ID and number of people linked to it
      result = await tx.run(`
        MATCH (o:Organization)
        RETURN o.id AS id, COUNT{(p:Person)-[r:WORKS_FOR]->(o)} AS employeesN
        ORDER BY o.createdDate DESC
        LIMIT 1
      `)
      if(result.records.length > 0) {
        orgInfo = result.records[0]
      }

      if(orgInfo != undefined && orgInfo['employeesN'] == 0) {
        throw new Error('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 that
      if(orgInfo != undefined && orgInfo['employeesN'] < employeeThreshold) {
        result = await tx.run(`
          MATCH (o:Organization {id: $orgId})
          MATCH (p:Person {name: $name})
          MERGE (p)-[r:WORKS_FOR]->(o)
          RETURN $orgId AS id
          `, { orgId: orgInfo['id'], name: name }
        )

      // Otherwise, create a new Organization and link Person to it
      } else {
        result = await tx.run(`
          MATCH (p:Person {name: $name})
          CREATE (o:Organization {id: randomuuid(), createdDate: datetime()})
          MERGE (p)-[r:WORKS_FOR]->(o)
          RETURN o.id AS id
          `, { name: name }
        )
      }

      // Return the Organization ID to which the new Person ends up in
      return result.records[0].get('id')
    })
    console.log(`User ${name} added to organization ${orgId}`)
  }
  await session.close()
  await driver.close()
})()
```

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 should produce the same effect when run several times (*idempotent*)**, 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 transaction 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**. This means that a query must be completed before the next one can run, and it is the reason why the previous examples all use the `async/await` syntax. To maintain multiple concurrent transactions, see how to run asynchronous queries.

Run an explicit transaction

You can achieve full control over transactions by manually beginning one with the method `Session.beginTransaction()`. You run queries inside an explicit transaction with the method `Transaction.run()`, as you do in transaction functions.

```
let session = driver.session({ database: 'neo4j' })
let transaction = await session.beginTransaction()

// use tx.run() to run queries
//     tx.commit() to commit the transaction
//     tx.rollback() to rollback the transaction

await transaction.commit()
await session.close()
```

An explicit transaction can be committed with `Transaction.rollback()`. If no explicit action is taken, the driver will automatically roll 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.

Example stub with an explicit transaction involving external APIs

```
const neo4j = require('neo4j-driver');
const URI = '<URI for Neo4j database>';
const USER = '<Username>';
const PASSWORD = '<Password>';

(async () => {

  try {
    driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
    await driver.verifyConnectivity()
  } catch(err) {
    console.log(`-- Connection error --\n${err}\n-- Cause --\n${err.cause}`)
    await driver.close()
    return
  }

  let customerId = await createCustomer(driver)
  let otherBankId = 42
  await transferToOtherBank(driver, customerId, otherBankId, 999)
  await driver.close()
})()

async function createCustomer(driver) {
  let { records } = await driver.executeQuery(`
      MERGE (c:Customer {id: randomUUID()})
      RETURN c.id AS id
    `, {},
    { database: 'neo4j' }
  )
  return records[0].get("id")
}

async function transferToOtherBank(driver, customerId, otherBankId, amount) {
  const session = driver.session({ database: 'neo4j' })
  const tx = await session.beginTransaction()
  try {
    if(! checkCustomerBalance(tx, customerId, amount))
      return

    try {
      decreaseCustomerBalance(tx, customerId, amount)
      await tx.commit()
```

```
    } catch (error) {
      requestInspection(customerId, otherBankId, amount, e)
      throw error  // roll back
    }

    await otherBankTransferApi(customerId, otherBankId, amount)
    // Now the money has been transferred => can't rollback anymore
    // (cannot rollback external services interactions)
  } finally {
    await session.close()
  }
}

async function checkCustomerBalance(tx, customerId, amount) {
  result = await tx.run(`
    MATCH (c:Customer {id: $id})
    RETURN c.balance >= $amount AS sufficient
    `, { id: customerId, amount: amount },
    { database: 'neo4j' }
  )
  return result.records[0].get('sufficient')
}

async function otherBankTransferApi(customerId, otherBankId, amount) {
  // make some API call to other bank
}

async function decreaseCustomerBalance(tx, customerId, amount) {
  await tx.run(`
    MATCH (c:Customer {id: $id})
    SET c.balance = c.balance - $amount
    `, { id: customerId, amount: amount }
  )
}

async function requestInspection(customerId, otherBankId, amount, error) {
  // manual cleanup required; log this or similar
  console.log('WARNING: transaction rolled back due to exception:')
  console.log(error)
}
```

Session configuration

When creating a session, you can provide an optional parameter of type `SessionConfig` to specify session configuration values.

Database selection

You should **always specify the database explicitly** with the `database` parameter, 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 set in the Neo4j instance settings is used.

```
const session = driver.session({
  database: '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 `defaultAccessMode` parameter to either `neo4j.session.READ` or `neo4j.session.WRITE`. Note that `.executeRead()` and `.executeWrite()` automatically override the session's default access mode.

```
const session = driver.session({
  database: 'neo4j',
  defaultAccessMode: neo4j.session.READ
})
```

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 are routed to any node of a cluster, whereas *write* ones are 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 parameter `impersonatedUser`, specifying the name of the user to impersonate. For this to work, the user under which the `Driver` was created needs to have the appropriate permissions. Impersonating a user is cheaper than creating a new `Driver` object.

```
const session = driver.session({
    database: '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 a second optional parameter of type `TransactionConfig` to `.executeRead()`, `.executeWrite()`, and `.beginTransaction()`. You can specify:

- A transaction timeout (in milliseconds). 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.
- An object 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 YIELD *` Cypher command. Use this to tag transactions.

```
let session = driver.session({ database: 'neo4j' })
const people_n = await session.executeRead(
  async tx => { return await tx.run('MATCH (a:Person) RETURN count(a)') },
  { timeout: 5000, metadata: {'app_name': 'people'} }  // TransactionConfig
)
```

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 important to always close sessions when you are done with them, so that they can be returned to the connection pool to be later reused. The best way is to wrap session usage in a `try/finally` block, calling `session.close()` in the `finally` clause.

```
let session = driver.session({database: 'neo4j'})
try {
  // use session to run queries
} finally {
  await session.close()
}
```

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 `Driver.executeQuery()`, the execution summary is part of the default return object, under the `summary` attribute.

```
let { records, summary } = await driver.executeQuery(`
  UNWIND ['Alice', 'Bob'] AS name
  MERGE (p:Person {name: name})
  `, {},
  { database: 'neo4j' }
)
// or summary = (await driver.executeQuery('<QUERY>')).summary
```

If you are using transaction functions, you can retrieve the query execution summary with the attribute `Result.summary` if you used `await` to resolve the result promise, or with the method `Result.summary()` if you consumed the result as a promise.

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.

```
let session = driver.session({ database: 'neo4j' })
try {
  let summary = await session.executeWrite(async tx => {
    let result = await tx.run(`
        UNWIND ['Alice', 'Bob'] AS name
        MERGE (p:Person {name: name})
    `)
    return result.summary
    // or result.summary(), if you don't await tx.run()
  })
} finally {
  session.close()
}
```

Query counters

The property `ResultSummary.counters` contains counters for the operations that a query triggered (as a `QueryStatistics` object).

Insert some data and display the query counters

```
let { records, summary } = await driver.executeQuery(`
  UNWIND ['Alice', 'Bob'] AS name
  MERGE (p:Person {name: name})
  `, {},
  { database: 'neo4j' }
)
console.log(summary.counters.updates())
/*
{
  nodesCreated: 2,
  nodesDeleted: 0,
  relationshipsCreated: 0,
  relationshipsDeleted: 0,
  propertiesSet: 0,
  labelsAdded: 1,
  labelsRemoved: 0,
  indexesAdded: 0,
  indexesRemoved: 0,
  constraintsAdded: 0,
  constraintsRemoved: 0
}
*/
console.log(summary.counters.containsUpdates())  // true
console.log(summary.counters.containsSystemUpdates())  // false
```

There are two additional methods on `ResultSummary.counters` 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. The plan is then available as a 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).

```
let result = await driver.executeQuery('EXPLAIN MATCH (p {name: $name}) RETURN p', { name: 'Alice' })
console.log(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. The plan is available under the property `ResultSummary.profile`. Notice that the query is also *run*, so the result object also contains any result records.

```
let result = await driver.executeQuery('PROFILE MATCH (p {name: $name}) RETURN p', { name: 'Alice' })
console.log(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 `minimumSeverityLevel` and/or `disabledCategories` /`disabledClassifications` 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` classifications

```
// at driver level
let driver = neo4j.driver(
  URI,  neo4j.auth.basic(USER, PASSWORD), {
    notificationsFilter: {
      minimumSeverityLevel: 'WARNING',  // or 'OFF' to disable entirely
      disabledClassifications: ['HINT', 'GENERIC']  // filters categories as well
    }
  }
)

// at session level
let session = driver.session({
    database: 'neo4j',
    notificationsFilter: {
      minimumSeverityLevel: 'WARNING',  // or 'OFF' to disable entirely
      disabledClassifications: ['HINT', 'GENERIC']  // filters categories as well
    }
})
```

Run non-blocking asynchronous queries

The examples in Query the database use the `async/await` syntax, which forces the driver to work synchronously. When using `await` with a query, your application waits for the server to retrieve all query results and transmit them to the driver. This is not a problem for most use cases, but for queries that have a long processing time or a large result set, asynchronous handling may speed up your application.

There are several ways of running asynchronous queries:

- **Asynchronous iteration** — the query result is processed (iteratively) as quickly as your application can handle. The driver modulates the amount of records transmitted by the server accordingly.
- **Promise API** — the query result is returned as a `Promise`. The promise is only resolved when the full result set is available to the driver. Best suited for queries with a large server processing time, but the result of which you want to process in one go. Your application receives the result in bulk, for eager consumption.
- **Streaming API** — the query result is returned as a stream, so that each result record is processed as soon as it is available. Best suited for queries where records processing is individual. Your application receives the result in bits, for lazy consumption.
- **Reactive API** — for reactive applications.

When using `await tx.run()` in a transaction function, you may return the query result out of the transaction function as-is for further processing. On the other hand, **for async queries, you have to process the result *inside* the transaction function** (except for the Promise API).

Asynchronous iteration

The `Result` object supports asynchronous iteration. This allows your application to process data at its own pace, with the driver accordingly modulating the speed at which records are streamed from the server, applying backpressure. With async iterators, you get the guarantee that your application does not receive data faster than it can process.

```
const session = driver.session()
try {
  const peopleNames = await session.executeWrite(async tx => {
    const result = tx.run(  ①
      'MERGE (p:Person {name: $name}) RETURN p.name AS name',
      { name: 'Alice' }
    )
    let names = []
    for await (const record of result) {  ②
      console.log(`Processing ${record.get('name')}`)
      names.push(record.get('name'))
    }
    return names  ③
  })
} finally {
  await session.close()
}
```

① Run a query

② Process records with async iteration

③ Return *processed* results (and not the raw query result)

There are two important points to the usage of the async iterator:

- **you can async-iterate only *once* per query result**. Once the result *cursor* reaches the end of the stream, it does not rewind, so you cannot iterate over a result more than once. If you need to process the data more than once in your application, you have to manually store it in an auxiliary data structure (like a list, as above).
- **the processing of the result happens *inside* the** transaction function. You **should not** return the raw result out of the transaction function and *then* iterate over it. That workflow only works with the Promise API.

Promise API

The Promise API allows to run a query and receive the result as a `Promise`. You can think of this query method as allowing you to specify a Cypher query and a number of callbacks that are asynchronously executed depending on the query outcome.

```
const session = driver.session({database: 'neo4j'})
const result = session.executeWrite(async tx => {  ①
  return tx.run(
    'MERGE (p:Person {name: $name}) RETURN p.name AS name',
    { name: 'Alice' }
  )
})
result.then(result => {  ②
  result.records.forEach(record => {
    console.log(record.get('name'))
  })
  return result
})
.catch(error => {  ③
  console.log(error)
})
.then(() => session.close())  ④
```

① Run a query

② Specify callback for successful runs, taking query result as input

③ Specify callback for failed runs, taking driver error as input

④ Specify callback to run regardless of query outcome

The Promise API holds for `Driver.executeQuery()` as well. If you prepend `await` to an `.executeQuery()` call, as shown in Query the database, you effectively force your application to wait until the result is ready, and you obtain the corresponding of either `result` or `error` from the example above.

Combine multiple transactions

To run multiple queries within the same transaction, use `Promise.all()`. It runs asynchronous operations concurrently, so you can submit multiple queries at the same time and wait for them all to finish.

Retrieve people's names and assign each of them to the company Neo4j

```
const companyName = 'Neo4j'
const session = driver.session({database: 'neo4j'})
try {
  const names = await session.executeRead(async tx => {
    const result = await tx.run('MATCH (p:Person) RETURN p.name AS name')
    return result.records.map(record => record.get('name'))
  })

  const relationshipsCreated = await session.executeWrite(tx =>
    Promise.all(  // group together all Promises
      names.map(name =>
        tx.run(`
          MATCH (emp:Person {name: $personName})
          MERGE (com:Company {name: $companyName})
          MERGE (emp)-[:WORKS_FOR]->(com)
          `, { personName: name, companyName: companyName }
        )
        .then(result => result.summary.counters.updates().relationshipsCreated)
      )
    ).then(values => values.reduce((a, b) => a + b))  // aggregate results
  )

  console.log(`Created ${relationshipsCreated} employees relationships.`)
} finally {
  await session.close()
}
```

Streaming API

The Streaming API allows to run a query and receive results individually, as soon as the server has them ready. You can specify a callback to process each record. This API is particularly fit for cases in which it may take the server a different time to retrieve the different records, but you want to process each of them as soon as they are available. The behavior is similar to the async iterator; the programming style is different.

```
const session = driver.session({database: 'neo4j'})
let peopleNames = []

session
  .run('MERGE (p:Person {name: $name}) RETURN p.name AS name', {  ①
    name: 'Alice'
  })
  .subscribe({  ②
    onKeys: keys => {  ③
      console.log('Result columns are:')
      console.log(keys)
    },
    onNext: record => {  ④
      console.log(`Processing ${record.get('name')}`)
      peopleNames.push(record.get('name'))
    },
    onCompleted: () => {  ⑤
      session.close() // returns a Promise
    },
    onError: error => {  ⑥
      console.log(error)
    }
  })
```

① Run a query

② Attach a handler to the result stream

③ The onKeys callback receives the list of result columns

④ The `onNext` callback is invoked every time a record is received

⑤ The `onCompleted` callback is invoked when the transaction is over

⑥ The `onError` is triggered in case of error

Reactive API

Typical of reactive programming, in a reactive flow consumers control the rate at which they consume records from queries, and the driver in turn manages the rate at which records are requested from the server. The reactive API is recommended for applications that are already oriented towards the reactive style.

```
const rxjs = require('rxjs');

const rxSession = driver.rxSession()  ①
const rxResult = await rxSession.executeWrite(tx => {
  return tx
    .run('MERGE (p:Person {name: $name}) RETURN p.name AS name', {  ②
      name: 'Alice'
    })
    .records()  ③
    .pipe(  ④
      rxjs.map(record => record.get('name')),
      //rxjs.materialize(),  // optional, turns outputs into Notifications
      rxjs.toArray()
    )
  })
const people = await rxResult.toPromise()
console.log(people)
```

① Obtain a reactive session

② Run a query

③ Obtain an observable for result records

④ Reactive processing

The reactive API requires a `RxResult`.

The reactive API is not available in the lite version of the driver.

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.

```
await driver.executeQuery('<QUERY 1>')

// subsequent executeQuery calls will be causally chained

await driver.executeQuery('<QUERY 2>') // can read result of <QUERY 1>
await driver.executeQuery('<QUERY 3>') // can read result of <QUERY 2>
```

To disable bookmark management and causal consistency, set the `bookmarkManager` option to `null` in `.executeQuery()` calls.

```
await driver.executeQuery(
    '<QUERY>',
    {},
    {
      bookmarkManager: null
    }
)
```

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.

```
let session = driver.session({database: 'neo4j'})
try {
  await session.executeWrite(async tx => {
    await tx.run("<QUERY 1>")
  })
  await session.executeWrite(async tx => {
    await tx.run("<QUERY 2>")  // can read result of QUERY 1
  })
  await session.executeWrite(async tx => {
    await tx.run("<QUERY 3>")  // can read result of QUERY 1, 2
  })
} finally {
  await session.close()
}
```

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.

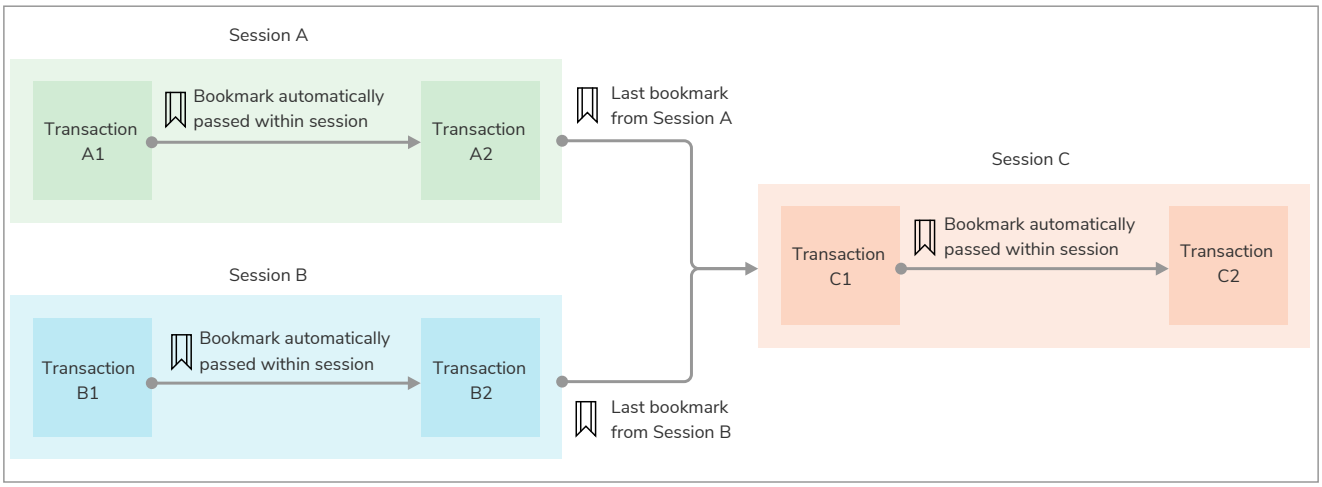
Coordinate multiple sessions using bookmarks

```
const neo4j = require('neo4j-driver');

(async () => {
  const URI = '<URI to Neo4j database>'
  const USER = '<Username>'
  const PASSWORD = '<Password>'
  let driver
  try {
    driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
    await driver.verifyConnectivity()
  } catch(err) {
    console.log(`-- Connection error --\n${err}\n-- Cause --\n${err.cause}`)
    return
  }
  await createFriends(driver)
})()

async function createFriends(driver) {
  let savedBookmarks = []  // To collect the sessions' bookmarks

  // Create the first person and employment relationship.
  const sessionA = driver.session({database: 'neo4j'})
  try {
    await createPerson(sessionA, 'Alice')
    await employPerson(sessionA, 'Alice', 'Wayne Enterprises')
    savedBookmarks.concat(sessionA.lastBookmarks())  ①
  } finally {
    sessionA.close()
  }

  // Create the second person and employment relationship.
  const sessionB = driver.session({database: 'neo4j'})
  try {
    await createPerson(sessionB, 'Bob')
    await employPerson(sessionB, 'Bob', 'LexCorp')
    savedBookmarks.concat(sessionB.lastBookmarks())  ①
  } finally {
    sessionB.close()
  }

  // Create (and show) a friendship between the two people created above.
  const sessionC = driver.session({
```

```
    database: 'neo4j',
    bookmarks: savedBookmarks  ②
  })
  try {
    await createFriendship(sessionC, 'Alice', 'Bob')
    await printFriendships(sessionC)
  } finally {
    sessionC.close()
  }
}

// Create a person node.
async function createPerson(session, name) {
  await session.executeWrite(async tx => {
    await tx.run('CREATE (:Person {name: $name})', { name: name })
  })
}

// Create an employment relationship to a pre-existing company node.
// This relies on the person first having been created.
async function employPerson(session, personName, companyName) {
  await session.executeWrite(async tx => {
    await tx.run(`
      MATCH (person:Person {name: $personName})
      MATCH (company:Company {name: $companyName})
      CREATE (person)-[:WORKS_FOR]->(company)`,
      { personName: personName, companyName: companyName }
    )
  })
}

// Create a friendship between two people.
async function createFriendship(session, nameA, nameB) {
  await session.executeWrite(async tx => {
    await tx.run(`
      MATCH (a:Person {name: $nameA})
      MATCH (b:Person {name: $nameB})
      MERGE (a)-[:KNOWS]->(b)
      `, { nameA: nameA, nameB: nameB }
    )
  })
}

// Retrieve and display all friendships.
async function printFriendships(session) {
  const result = await session.executeRead(async tx => {
    return await tx.run('MATCH (a)-[:KNOWS]->(b) RETURN a.name, b.name')
  })
  for(record of result.records) {
    console.log(`${record.get('a.name')} knows ${record.get('b.name')}`)
  }
}
```

① Collect and combine bookmarks from different sessions using the method `Session.lastBookmarks()`.

② Use them to initialize another session with the `bookmarks` 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 option `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.

```
await driver.executeQuery('<QUERY 1>')

session = driver.session({
    bookmarkManager: driver.executeQueryBookmarkManager
})
try {
  // every query inside this session will be causally chained
  // (i.e., can read what was written by <QUERY 1>)
  await session.executeWrite(async tx => tx.run('<QUERY 2>'))
} finally {
  await session.close()
}

// subsequent executeQuery calls will be causally chained
// (i.e., can read what was written by <QUERY 2>)
await driver.executeQuery('<QUERY 3>')
```

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 `Driver.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 `Result` object.

```
let session = driver.session({database: 'neo4j'})
try {
  const result = await session.run(
    'MERGE (a:Person {name: $name})',
    { name: 'Alice'}
  )
} finally {
  await session.close()
}
```

Since the driver cannot figure out whether the query in a `session.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 keyword argument `defaultAccessMode=neo4j.session.READ` when creating the session.

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

You can run an implicit transaction with a reactive session as well.

Import CSV files

The most common use case for using `Session.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

```
let session = driver.session({database: 'neo4j'})
try {
  let result = await session.run(`
    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
  `)
  console.log(result.summary.counters.updates())
} finally {
  await session.close()
}
```

While `LOAD CSV` can be a convenience, there is nothing *wrong* in deferring the parsing of the CSV file to your JavaScript 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 an optional third parameter of type `TransactionConfig` to `session.run()`. The configuration allows to specify a query timeout and to attach metadata to the transaction. For more information, see Transactions — Transaction configuration.

```
let session = driver.session({database: 'neo4j'})
let result = await session.run(
  'MATCH (a:Person) RETURN count(a) AS people',
  {},  // query parameters
  { timeout: 5000, metadata: {'appName': 'peopleTracker'} }  // transactionConfig
)
```

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

```
let dangerousLabel = 'Special Person\\u0060'
// convert \u0060 to literal backtick, then escape backticks
let escapedLabel = dangerousLabel.replace(/\\u0060/g, '``').replace(/`/g, '```')

let result = await driver.executeQuery(
  'MATCH (p:`' + escapedLabel + '`) RETURN p.name',
  {},
  {database: 'neo4j'}
)
console.log(`Executed query: ${result.summary.query.text}`)
```

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

```
let propertyKey = 'name'
let label = 'Person'
let result = await driver.executeQuery(
  'CALL apoc.merge.node($labels, $properties)',
  {labels: [label], properties: {property_key: 'Alice'}},
  {database: 'neo4j'}
)
```

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

Logging

When creating a `Driver` instance, you may optionally specify its logging configuration.

Logging is off by default. To turn it on, specify the `logging` option when you initialize the driver. As value, use the function `neo4j.logging.console()`, which logs to console and takes an optional parameter `level`. The logging level can be either `error`, `warn`, `info`, or `debug`. Enabling one level automatically enables all higher priority ones, and its default is `info`.

Creating a driver with logging of level `debug` to console

```
neo4j.driver(
  URI,
  neo4j.auth.basic(USER, PASSWORD),
  {  // driver config
    logging: neo4j.logging.console('debug')
  }
)
```

Example of log output upong driver connection

```
1681215847749 INFO Routing driver 0 created for server address localhost:7687
1681215847765 INFO Routing table is stale for database: "neo4j" and access mode: "WRITE":
RoutingTable[database=neo4j, expirationTime=0, currentTime=1681215847765, routers=[], readers=[],
writers=[]]
1681215847773 DEBUG Connection [0][] created towards localhost:7687(127.0.0.1)
1681215847773 DEBUG Connection [0][] C: HELLO {user_agent: 'neo4j-javascript/5.3.0', ...}
1681215847778 DEBUG Connection [0][] S: SUCCESS
{"signature":112,"fields":[{"server":"Neo4j/5.8.0","connection_id":"bolt-
1782","hints":{"connection.recv_timeout_seconds":{"low":120,"high":0}}}]}
1681215847778 DEBUG Connection [0][bolt-1782] created for the pool localhost:7687
1681215847778 DEBUG Connection [0][bolt-1782] acquired from the pool localhost:7687
1681215847779 DEBUG Connection [0][bolt-1782] C: ROUTE {"address":"localhost:7687"} [] {"db":"neo4j"}
1681215847781 DEBUG Connection [0][bolt-1782] S: SUCCESS
{"signature":112,"fields":[{"rt":{"servers":[{"addresses":["localhost:7687"],"role":"WRITE"},{"addresses":
["localhost:7687"],"role":"READ"},{"addresses":["localhost:7687"],"role":"ROUTE"}],"ttl":{"low":300,"high"
:0},"db":"neo4j"}}]}
```

You may also specify a custom logger function, for example to log to a file. In that case, the `logging` option expects two properties:

- `level`: either `error`, `warn`, `info`, or `debug`. Enabling one level automatically enables all higher priority ones. Defaults to `info`.
- `logger`: a function invoked when a message needs to be logged. Takes `level, message` as input.

Creating a driver with logging of level `error` to console

```
neo4j.driver(
  URI,
  neo4j.auth.basic(USER, PASSWORD),
  {  // driver config
    logging: {
      level: 'error',
      logger: (level, message) => console.log(level + ' ' + message)
    }
  }
)
```

You can find more information about logging in the API documentation.

Performance recommendations

Always specify the target database

Specify the target database on all queries using the `database` parameter, both in `Driver.executeQuery()` calls and 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 them.

Good practices

```
await driver.executeQuery('<QUERY>', {}, {database: '<DB NAME>'})
```

```
driver.session({database: '<DB NAME>'})
```

Bad practices

```
await driver.executeQuery('<QUERY>')
```

```
driver.session()
```

Be aware of the cost of transactions

When submitting queries through `.executeQuery()` or through `.executeRead/Qrite()`, 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

```
let session = driver.session({database: '<DB NAME>'})
for(let i=0; i<1000; i++) {
    await session.run("<QUERY>")
}
session.close()
```

Privilege data integrity over throughput

```
for(let i=0; i<1000; i++) {
    await driver.executeQuery("<QUERY>", {}, {database: '<DB NAME>'})
    // or session.executeRead/Write() calls
}
```

Route read queries to cluster readers

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

- setting the `routing: READ` as configuration in `Driver.executeQuery()` calls
- using `Session.executeRead()` instead of `Session.executeWrite()` (for managed transactions)
- setting `AccessMode: neo4j.AccessModeRead` when creating a new session (for explicit transactions).

Good practices

```
await driver.executeQuery(
  'MATCH (p:Person) RETURN p.name',
  {},
  {
    routing: 'READ',  // short for neo4j.routing.READ
    database: 'neo4j'
  }
)
```

```
let session = driver.session({ database: 'neo4j' })
await session.executeRead(async tx => {
  return await tx.run('MATCH (p:Person) RETURN p.name', {})
})
```

Bad practices

```
await driver.executeQuery(
  'MATCH (p:Person) RETURN p.name',
  {},
  {
    database: 'neo4j'
  }
)
// defaults to routing = writers
```

```
let session = driver.session({ database: 'neo4j' })
await session.executeRead(async tx => {
  return await tx.run('MATCH (p:Person) RETURN p.name', {})
})
// don't ask to write on a read-only operation
```

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 function, for both nodes and relationships.

```
// Create an index on Person.name
await driver.executeQuery('CREATE INDEX personName FOR (n:Person) ON (n.name)')
```

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 in the `profile` property of the `ResultSummary` object.

```
const result = await driver.executeQuery('PROFILE MATCH (p {name: $name}) RETURN p', { name: 'Alice' })
console.log(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 in the `plan` property of the `ResultSummary` object.

```
const result = await driver.executeQuery('EXPLAIN MATCH (p {name: $name}) RETURN p', { name: 'Alice' })
console.log(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. To learn how to combine labels, see Cypher → Label expressions.

Good practices

```
await driver.executeQuery(
  'MATCH (p:Person|Animal {name: $name}) RETURN p',
  { name: 'Alice' }
)
```

```
let session = driver.session({database: '<DB NAME>'})
await session.run(
  'MATCH (p:Person|Animal {name: $name}) RETURN p',
  { name: 'Alice' }
)
```

Bad practices

```
await driver.executeQuery(
  'MATCH (p {name: $name}) RETURN p',
  { name: 'Alice' }
)
```

```
let session = driver.session({database: '<DB NAME>'})
await session.run(
  'MATCH (p {name: $name}) RETURN p',
  { name: 'Alice' }
)
```

Batch data creation

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

Good practice

```
numbers = []
for(let i=0; i<10000; i++) {
  numbers.push({value: Math.random()})
}
await driver.executeQuery(`
    WITH $numbers AS batch
    UNWIND batch AS node
    MERGE (n:Number {value: node.value})
    `, { numbers: numbers }
)
```

Bad practice

```
for(let i=0; i<10000; i++) {
  await driver.executeQuery(
    'MERGE (:Number {value: $value})',
    { value: Math.random() }
  )
}
```

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

Use query parameters instead of hardcoding or concatenating values into queries. This allows to leverage the database's query cache.

Good practices

```
await driver.executeQuery(
  'MATCH (p:Person {name: $name}) RETURN p',
  { name: 'Alice' }  // query parameters
)
```

```
let session = driver.session({database: '<DB NAME>'})
await session.run(
  'MATCH (p:Person {name: $name}) RETURN p',
  { name: 'Alice' }  // query parameters
)
```

Bad practices

```
await driver.executeQuery('MATCH (p:Person {name: "Alice"}) RETURN p')

let name = "Alice"
await driver.executeQuery('MATCH (p:Person {name: "' + name + '"}) RETURN p')
```

```
let session = driver.session({database: '<DB NAME>'})
await session.run(
  'MATCH (p:Person {name: "Alice"}) RETURN p',
  // or 'MATCH (p:Person {name: ' + name + '}) RETURN p'
  {}
)
```

Concurrency

Use asynchronous querying. 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 `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.

Usage within a browser (WebSockets)

There is a special browser version of the driver, which supports connecting to Neo4j over WebSockets. You can include it in an HTML page using one of the following tags:

```
<!-- Direct reference -->
<script src="lib/browser/neo4j-web.min.js"></script>

<!-- unpkg CDN non-minified -->
<script src="https://unpkg.com/neo4j-driver"></script>
<!-- unpkg CDN minified for production use, version X.Y.Z -->
<script src="https://unpkg.com/neo4j-driver@X.Y.Z/lib/browser/neo4j-web.min.js"></script>

<!-- jsDelivr CDN non-minified -->
<script src="https://cdn.jsdelivr.net/npm/neo4j-driver"></script>
<!-- jsDelivr CDN minified for production use, version X.Y.Z -->
<script src="https://cdn.jsdelivr.net/npm/neo4j-driver@X.Y.Z/lib/browser/neo4j-web.min.js"></script>
```

Including the driver in the page makes a global `neo4j` object available, through which you may then connect and query a Neo4j database, in the same way as you would do for a local application.

Driver usage example

```
const URI = '<URI for Neo4j database>'
const USER = '<Username>'
const PASSWORD = '<Password>'
const driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
const serverInfo = await driver.getServerInfo()

// use driver to run queries

await driver.close()
```

Code running in a browser is visible to the client, **including your database credentials**. You have to take extra steps to ensure that no unauthorized access is allowed.

Starting from version 5.4, the browser version is also exported as an ECMA Script Module. You can import it using one of the following statements:

```
// Direct reference
import neo4j from 'lib/browser/neo4j-web.esm.min.js'

// unpkg CDN non-minified , version X.Y.Z where X.Y.Z >= 5.4.0
import neo4j from 'https://unpkg.com/neo4j-driver@X.Y.Z/lib/browser/neo4j-web.esm.js'

// unpkg CDN minified for production use, version X.Y.Z where X.Y.Z >= 5.4.0
import neo4j from 'https://unpkg.com/neo4j-driver@X.Y.Z/lib/browser/neo4j-web.esm.min.js'

// jsDelivr CDN non-minified, version X.Y.Z where X.Y.Z >= 5.4.0
import neo4j from 'https://cdn.jsdelivr.net/npm/neo4j-driver@X.Y.Z/lib/browser/neo4j-web.esm.js'

// jsDelivr CDN minified for production use, version X.Y.Z where X.Y.Z >= 5.4.0
import neo4j from 'https://cdn.jsdelivr.net/npm/neo4j-driver@X.Y.Z/lib/browser/neo4j-web.esm.min.js'
```

It is not required to explicitly close the driver on a web page, unless its lifetime does not correspond to the lifetime of the page. The reason for this is that web browsers are supposed to gracefully terminate all open WebSockets when a page is closed.

=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.

```
const driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
```

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 const driver = neo4j.driver(URI, neo4j.auth.kerberos(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 const driver = neo4j.driver(URI, neo4j.auth.bearer(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

If the server is equipped with a custom authentication scheme, use `neo4j.auth.custom`.

```
1 const driver = neo4j.driver(
2   URI,
3   neo4j.auth.custom(principal, credentials, realm, scheme, parameters)
4 )
```

No authentication

If authentication is disabled on the server, the authentication parameter can be omitted entirely.

Custom address resolver

When creating a `Driver` 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.

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

```
let URI = 'neo4j://example.com:9999'
let addresses = [
  'localhost:7687'
]
let driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD), {
  resolver: address => addresses
})
```

Further connection parameters

You can find all `Driver` configuration parameters in the API documentation.

Data types and mapping to Cypher types

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

Core types

Cypher Type	JavaScript Type
NULL	`null`
LIST	`Array`
MAP	`Object`
BOOLEAN	`Boolean`
INTEGER	`Integer`
FLOAT	`Number`
STRING	`String`
ByteArray	`Int8Array`

`Integer` is not one of JavaScript's native types, but rather a custom one accomodating Cypher's precision. You can disable this through the `disableLosslessIntegers` configuration entry when instantiating the driver, so that JavaScript's native `Number` type is used instead. Note that this can lead to a loss of precision.

Temporal types

Temporal data types are ISO-8601-compliant. To serialize them to string, use the `.toString()` method. Temporal objects are immutable.

Sub-second values are measured to nanosecond precision. To convert between driver and native types, use the methods `.fromStandardDate()` and `.toStandardDate()` (does not apply to `Duration`). Since JavaScript date types do not support nanoseconds, `.fromStandardDate()` allows a nanoseconds argument (optional), and `.toStandardDate()` drops the nanoseconds.

For a list of time zone abbreviations, see List of tz database time zones.

Cypher Type	JavaScript Type
DATE	`Date`
ZONED TIME	`Time`
LOCAL TIME	`LocalTime`
ZONED DATETIME	`DateTime`
LOCAL DATETIME	`LocalDateTime`
DURATION	`Duration`

How to use temporal types in queries

```
const neo4j = require('neo4j-driver');
const URI = '<URI for Neo4j database>';
const USER = '<Username>';
const PASSWORD = '<Password>';

(async () => {
  const driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))

  const friendsSince = new neo4j.types.DateTime(1999, 11, 23, 7, 47, 0, 4123, -4*3600, 'Europe/Berlin')

  // JS native types work as well.
  // They don't support the full feature-set of Neo4j's type though.
  // let jsFriendsSince = new Date(1999, 11, 23, 7, 47, 0)
  // jsFriendsSince = new neo4j.types.Date.fromStandardDate(jsFriendsSince)

  // Create a friendship with the given DateTime, and return the DateTime
  const result = await driver.executeQuery(`
    MERGE (a:Person {name: $name})
    MERGE (b:Person {name: $friend})
    MERGE (a)-[friendship:KNOWS]->(b)
    SET friendship.since = $friendsSince
    RETURN friendship.since
    `, {
      name: 'Alice', friend: 'Bob',
      friendsSince: friendsSince  // or friendsSince: jsFriendsSince
    }
  )
  const outDateTime = result.records[0].get('friendship.since')
  console.log(outDateTime)
  /*
  DateTime {
    year: Integer { low: 1999, high: 0 },
    month: Integer { low: 11, high: 0 },
    day: Integer { low: 23, high: 0 },
    hour: Integer { low: 6, high: 0 },
    minute: Integer { low: 47, high: 0 },
    second: Integer { low: 0, high: 0 },
    nanosecond: Integer { low: 4123, high: 0 },
    timeZoneOffsetSeconds: Integer { low: -18000, high: -1 },
    timeZoneId: 'Europe/Berlin'
  }
 */

  // Convert DateTime to JS native Date (lossy)
  const jsOutDateTime = outDateTime.toStandardDate()
  console.log(jsOutDateTime)
  // 1999-11-23T11:47:00.000Z

  await driver.close()
})()
```

Date

Represents an instant capturing the date, but not the time, nor the timezone.

```
d = new neo4j.Date(2021, 11, 2)
// Date { year: 2021, month: 11, day: 2 }
d.toString() // '2021-11-02'
```

For full documentation, see API documentation — Date.

Time

Represents an instant capturing the time, and the timezone offset in seconds, but not the date.

```
d = new neo4j.Time(7, 47, 0, 4123, -4*3600)
/*
Time {
  hour: 7,
  minute: 47,
  second: 0,
  nanosecond: 4123,
  timeZoneOffsetSeconds: -14400
}
*/
d.toString() // '07:47:00.000004123-04:00'
```

For full documentation, see API documentation — Time.

LocalTime

Represents an instant capturing the time of day, but not the date, nor the timezone.

```
d = new neo4j.LocalTime(7, 47, 0, 4123)
// LocalTime { hour: 7, minute: 47, second: 0, nanosecond: 4123 }
d.toString() // '07:47:00.000004123'
```

For full documentation, see API documentation — LocalTime.

DateTime

Represents an instant capturing the date, the time, and the timezone identifier. Timezone parameters (offset and identifier) are optional.

```
d = new neo4j.DateTime(2021, 11, 2, 7, 47, 0, 4123, -4*3600, 'Europe/Berlin')
/*
DateTime {
  year: 2021,
  month: 11,
  day: 2,
  hour: 7,
  minute: 47,
  second: 0,
  nanosecond: 4123,
  timeZoneOffsetSeconds: -14400,
  timeZoneId: 'Europe/Berlin'
}
*/
d.toString() // '2021-11-02T07:47:00.000004123-04:00[US/Eastern]'
```

For full documentation, see API documentation — DateTime.

LocalDateTime

Represents an instant capturing the date and the time, but not the timezone.

```
d = new neo4j.LocalDateTime(2021, 11, 2, 7, 47, 0, 4123)
/*
LocalDateTime {
  year: 2021,
  month: 11,
  day: 2,
  hour: 7,
  minute: 47,
  second: 0,
  nanosecond: 4123
}
*/
d.toString() // '2021-11-02T07:47:00.000004123'
```

For full documentation, see API documentation — LocalDateTime.

Duration

Represents the difference between two points in time.

```
const d = new neo4j.Duration(1, 2, 3, 4)
/*
Duration {
  months: 1,
  days: 2,
  seconds: Integer { low: 3, high: 0 },
  nanoseconds: Integer { low: 4, high: 0 }
}
*/
d.toString() // 'P1M2DT3.000000004S'
```

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 driver has a single type `neo4j.types.Point`, which can behave as a 2D/3D cartesian/WGS-84 point, depending on the `SRID` it is initialized with. The `SRID` (short for *Spatial Reference Identifier*) is a number identifying the coordinate system the point is to be interpreted in. You can think of it as a unique identifier for each spatial type.

SRID	Description
7203	2D point in the cartesian space.
9157	3D point in the cartesian space.
4326	2D point in the WGS84 space.
4979	3D point in the WGS84 space.

Points in cartesian space

```
// A 2D Point in cartesian space
const point2d = new neo4j.types.Point(
  7203, // SRID
  1, // x
  5.1 // y
)
// Point { srid: 4979, x: 1, y: -2 }

// A 3D Point in cartesian space
const point3d = new neo4j.types.Point(
  9157, // SRID
  1, // x
  -2, // y
  3.1 // z
)
// Point { srid: 4979, x: 1, y: -2, z: 3.1 }
```

Points in WGS-84 space

```
// A 2D point in WGS-84 space
const point2d = new neo4j.types.Point(
  4326, // SRID
  1, // x
  -2, // y
  3.1 // z
)
// Point { srid: 4979, x: 1, y: -2}

// A 3D point in WGS-84 space
const point3d = new neo4j.types.Point(
  4979, // SRID
  1, // x
  -2, // y
  3.1 // z
)
// Point { srid: 4979, x: 1, y: -2, z: 3.1 }
```

For full documentation, see API documentation — Point.

Graph types

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

Cypher Type	JavaScript Type
NODE	`neo4j.types.Node`
RELATIONSHIP	`neo4j.types.Relationship`
PATH	`neo4j.types.Path`
	`neo4j.types.PathSegment`

Node

Represents a node in a graph.
The property `elementId` provides an 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.

```
const neo4j = require('neo4j-driver');
const URI = '<URI for Neo4j database>';
const USER = '<Username>';
const PASSWORD = '<Password>';

(async () => {
  const driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
  const result = await driver.executeQuery(
    'MERGE (p:Person {name: $name}) RETURN p AS person',
    { name: 'Alice' }
  )
  const node = result.records[0].get('person')
  console.log(node)
  /*
  Node {
    identity: Integer { low: 393, high: 0 },  // deprecated
    labels: [ 'Person' ],
    properties: { name: 'Alice' },
    elementId: '4:d6154461-ff34-42a9-b7c3-d32673913419:393'
  }
  */
  await driver.close()
})()
```

For full documentation, see API documentation — Node.

Relationship

Represents a relationship in a graph.
The property `elementId` provides an 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. The same applies to `startNodeElementId` and `endNodeElementId`.

```
const neo4j = require('neo4j-driver');
const URI = '<URI for Neo4j database>';
const USER = '<Username>';
const PASSWORD = '<Password>';

(async () => {
  const driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))
  const result = await driver.executeQuery(`
    MERGE (p:Person {name: $name})
    MERGE (friend:Person {name: $friend_name})
    MERGE (p)-[r:KNOWS]->(friend)
    SET r.status = $status, r.since = date()
    RETURN r AS friendship
    `, {
      name: 'Alice', status: 'BFF', friend_name: 'Bob'
    }
  )
  const relationship = result.records[0].get('friendship')
  console.log(relationship)
  /*
  Relationship {
    identity: Integer { low: 388, high: 0 },  // deprecated
    start: Integer { low: 393, high: 0 },  // deprecated
    end: Integer { low: 394, high: 0 },  // deprecated
    type: 'KNOWS',
    properties: {
      since: Date { year: [Integer], month: [Integer], day: [Integer] },
      status: 'BFF'
    },
    elementId: '5:d6154461-ff34-42a9-b7c3-d32673913419:388',
    startNodeElementId: '4:d6154461-ff34-42a9-b7c3-d32673913419:393',
    endNodeElementId: '4:d6154461-ff34-42a9-b7c3-d32673913419:394'
  }
  */
  await driver.close()
})()
```

For full documentation, see API documentation — Relationship.

Path, PathSegment

`Path` represents a path in a graph, while `PathSegment` represents its individual links.

```
const neo4j = require('neo4j-driver');
const URI = '<URI for Neo4j database>';
const USER = '<Username>';
const PASSWORD = '<Password>';

(async () => {
  const driver = neo4j.driver(URI, neo4j.auth.basic(USER, PASSWORD))

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

  // Follow :KNOWS relationships outgoing from Alice three times, return as path
  const result = await driver.executeQuery(`
    MATCH p = (:Person {name: $name})-[:KNOWS*3]->(:Person)
    RETURN p AS friendsChain
    `, {
      name: 'Alice'
    }
  )
  const path = result.records[0].get('friendsChain')
  console.log(path)
  /*
  Path {
    start: Node {
      identity: Integer { low: 393, high: 0 },
      labels: [ 'Person' ],
      properties: { name: 'Alice' },
      elementId: '4:d6154461-ff34-42a9-b7c3-d32673913419:393'
    },
    end: Node {
      identity: Integer { low: 396, high: 0 },
      labels: [ 'Person' ],
      properties: { name: 'Sara' },
      elementId: '4:d6154461-ff34-42a9-b7c3-d32673913419:396'
    },
    segments: [
      PathSegment {
        start: [Node],
        relationship: [Relationship],
        end: [Node]
      },
      PathSegment {
        start: [Node],
        relationship: [Relationship],
        end: [Node]
      },
      PathSegment {
        start: [Node],
        relationship: [Relationship],
        end: [Node]
      }
    ],
    length: 3
  }
  */
  await driver.close()
})()

async function addFriend(driver, name, status, friendName) {
  await driver.executeQuery(`
    MERGE (p:Person {name: $name})
    MERGE (friend:Person {name: $friendName})
    MERGE (p)-[r:KNOWS]->(friend)
    SET r.status = $status, r.since = date()
    `, {
      name: name, status: status, friendName: friendName
    }
  )
}
```

For full documentation, see PathSegment.

Errors

All errors raised by the driver are of type `Neo4jError`. 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. All driver's exception types implement the method `.isRetryable()`, which 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.

API documentation

=GraphAcademy courses=

Graph Data Modeling Fundamentals

Intermediate Cypher Queries

Building Neo4j Applications with Node.js

Building Neo4j Applications with TypeScript

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.