## INSERT

const jane = await User.create({ firstName: "Jane", lastName: "Doe" });

console.log("Jane's auto-generated ID:", jane.id);

* only fields

const user = await User.create({

username: 'alice123',

isAdmin: true

}, { fields: ['username'] });

## SELECT

const users = await User.findAll();

console.log(users.every(user => user instanceof User)); // true

console.log("All users:", JSON.stringify(users, null, 2));

only fields

Model.findAll({

attributes: ['foo', 'bar']

});

* Rename

Model.findAll({

attributes: ['foo', ['bar', 'baz'], 'qux']

});

SELECT foo, bar AS baz, qux FROM ...

* Usage sequelize.fn

Model.findAll({

attributes: [

'foo',

[sequelize.fn('COUNT', sequelize.col('hats')), 'n\_hats'],

'bar'

]

});

SELECT foo, COUNT(hats) AS n\_hats, bar FROM ...

* Include (findAll and …)

Model.findAll({

attributes: {

include: [

[sequelize.fn('COUNT', sequelize.col('hats')), 'n\_hats']

]

}

});

SELECT id, foo, bar, baz, qux, hats, COUNT(hats) AS n\_hats FROM ...

* Exclude (findAll without …)

Model.findAll({

attributes: { exclude: ['baz'] }

});

## WHERE

const { Op } = require("sequelize");

Post.findAll({

where: {

authorId: {

[Op.eq]: 2

}

}

});

// SELECT \* FROM post WHERE authorId = 2

where: {

authorId: 12

status: 'active'

}

// SELECT \* FROM post WHERE authorId = 12 AND status = 'active';

where: {

[Op.and]: [

{ authorId: 12 },

{ status: 'active' }

]

}

// SELECT \* FROM post WHERE authorId = 12 AND status = 'active';

* Where IN

where: {

id: [1,2,3] // Same as using `id: { [Op.in]: [1,2,3] }`

}

// SELECT ... FROM "posts" AS "post" WHERE "post"."id" IN (1, 2, 3);

## AND, OR combination

rank: {

[Op.or]: {

[Op.lt]: 1000,

[Op.eq]: null

}

},

// rank < 1000 OR rank IS NULL

createdAt: {

[Op.lt]: new Date(),

[Op.gt]: new Date(new Date() - 24 \* 60 \* 60 \* 1000)

}

// createdAt < [timestamp] AND createdAt > [timestamp]

[Op.or]: [

{

title: {

[Op.like]: 'Boat%'

}

},

{

description: {

[Op.like]: '%boat%'

}

}

]

// title LIKE 'Boat%' OR description LIKE '%boat%'

Delete WHERE

const { Op } = require("sequelize");

Post.destroy({

where: {

authorId: {

[Op.or]: [12, 13]

}

}

});

// DELETE FROM post WHERE authorId = 12 OR authorId = 13;

* Truncate

// Truncate the table

await User.destroy({

truncate: true

});

Update WHERE

// Change everyone without a last name to "Doe"

await User.update({ lastName: "Doe" }, {

where: {

lastName: null

}

});

## Query With Function

Post.findAll({

where: sequelize.where(sequelize.fn('char\_length', sequelize.col('content')), 7)

});

// SELECT ... FROM "posts" AS "post" WHERE char\_length("content") = 7

const { Op } = require("sequelize");

Post.findAll({

where: {

[Op.and]: [{ a: 5 }, { b: 6 }], // (a = 5) AND (b = 6)

[Op.or]: [{ a: 5 }, { b: 6 }], // (a = 5) OR (b = 6)

someAttribute: {

// Basics

[Op.eq]: 3, // = 3

[Op.ne]: 20, // != 20

[Op.is]: null, // IS NULL

[Op.not]: true, // IS NOT TRUE

[Op.or]: [5, 6], // (someAttribute = 5) OR (someAttribute = 6)

// Using dialect specific column identifiers (PG in the following example):

[Op.col]: 'user.organization\_id', // = "user"."organization\_id"

// Number comparisons

[Op.gt]: 6, // > 6

[Op.gte]: 6, // >= 6

[Op.lt]: 10, // < 10

[Op.lte]: 10, // <= 10

[Op.between]: [6, 10], // BETWEEN 6 AND 10

[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15

// Other operators

[Op.all]: sequelize.literal('SELECT 1'), // > ALL (SELECT 1)

[Op.in]: [1, 2], // IN [1, 2]

[Op.notIn]: [1, 2], // NOT IN [1, 2]

[Op.like]: '%hat', // LIKE '%hat'

[Op.notLike]: '%hat', // NOT LIKE '%hat'

[Op.startsWith]: 'hat', // LIKE 'hat%'

[Op.endsWith]: 'hat', // LIKE '%hat'

[Op.substring]: 'hat', // LIKE '%hat%'

[Op.iLike]: '%hat', // ILIKE '%hat' (case insensitive) (PG only)

[Op.notILike]: '%hat', // NOT ILIKE '%hat' (PG only)

[Op.regexp]: '^[h|a|t]', // REGEXP/~ '^[h|a|t]' (MySQL/PG only)

[Op.notRegexp]: '^[h|a|t]', // NOT REGEXP/!~ '^[h|a|t]' (MySQL/PG only)

[Op.iRegexp]: '^[h|a|t]', // ~\* '^[h|a|t]' (PG only)

[Op.notIRegexp]: '^[h|a|t]', // !~\* '^[h|a|t]' (PG only)

[Op.any]: [2, 3], // ANY ARRAY[2, 3]::INTEGER (PG only)

// In Postgres, Op.like/Op.iLike/Op.notLike can be combined to Op.any:

[Op.like]: { [Op.any]: ['cat', 'hat'] } // LIKE ANY ARRAY['cat', 'hat']

// There are more postgres-only range operators, see below

}

}

});

# Raw Query

const [results, metadata] = await sequelize.query("UPDATE users SET y = 42 WHERE x = 12");

* Results : là empty array
* Metadata : chứa numbers of affected rows.

\*với mysql, sqlserver 🡪 results, metadata là một

or

const { QueryTypes } = require('sequelize');

const users = await sequelize.query("SELECT \* FROM `users`", { type: QueryTypes.SELECT });

// We didn't need to destructure the result here - the results were returned directly

## Raw Query mapping 🡪 Model

const projects = await sequelize.query('SELECT \* FROM projects', {

model: Projects,

mapToModel: true // pass true here if you have any mapped fields

});

Template

const { QueryTypes } = require('sequelize');

await sequelize.query('SELECT 1', {

// A function (or false) for logging your queries

// Will get called for every SQL query that gets sent

// to the server.

logging: console.log,

// If plain is true, then sequelize will only return the first

// record of the result set. In case of false it will return all records.

plain: false,

// Set this to true if you don't have a model definition for your query.

raw: false,

// The type of query you are executing. The query type affects how results are formatted before they are passed back.

type: QueryTypes.SELECT

});

// Note the second argument being null!

// Even if we declared a callee here, the raw: true would

// supersede and return a raw object.

console.log(await sequelize.query('SELECT \* FROM projects', { raw: true }));

## Nested option

Without nest: true:

const { QueryTypes } = require('sequelize');

const records = await sequelize.query('select 1 as `foo.bar.baz`', {

type: QueryTypes.SELECT

});

Result

console.log(JSON.stringify(records[0], null, 2));

{

"foo.bar.baz": 1

}

With nest: true:

const { QueryTypes } = require('sequelize');

const records = await sequelize.query('select 1 as `foo.bar.baz`', {

nest: true,

type: QueryTypes.SELECT

});

Result

console.log(JSON.stringify(records[0], null, 2));

{

"foo": {

"bar": {

"baz": 1

}

}

}

## Replacements

const { QueryTypes } = require('sequelize');

await sequelize.query(

'SELECT \* FROM projects WHERE status = ?',

{

replacements: ['active'],

type: QueryTypes.SELECT

}

);

await sequelize.query(

'SELECT \* FROM projects WHERE status = :status',

{

replacements: { status: 'active' },

type: QueryTypes.SELECT

}

):

await sequelize.query(

'SELECT \* FROM projects WHERE status IN(:status)',

{

replacements: { status: ['active', 'inactive'] },

type: QueryTypes.SELECT

}

);

await sequelize.query(

'SELECT \* FROM users WHERE name LIKE :search\_name',

{

replacements: { search\_name: 'ben%' },

type: QueryTypes.SELECT

}

);

## Binding

Binding giống như replacements, nhưng binding sẽ được escaped trước khi sent query

const { QueryTypes } = require('sequelize');

await sequelize.query(

'SELECT \*, "text with literal $$1 and literal $$status" as t FROM projects WHERE status = $1',

{

bind: ['active'],

type: QueryTypes.SELECT

}

);

await sequelize.query(

'SELECT \*, "text with literal $$1 and literal $$status" as t FROM projects WHERE status = $status',

{

bind: { status: 'active' },

type: QueryTypes.SELECT

}

);

* If an array is passed, $1 is bound to the 1st element in the array (bind[0])
* If an object is passed, $key is bound to object['key'].
* **In either case $$ can be used to escape a literal $ sign.**