

**Version: v6 - stable**

Model Querying - Basics

Sequelize provides various methods to assist querying your database for data.

Important notice: to perform production-ready queries with Sequelize, make sure you have read the [Transactions guide](#) as well. Transactions are important to ensure data integrity and to provide other benefits.

This guide will show how to make the standard [CRUD](#) queries.

Simple INSERT queries

First, a simple example:

```
// Create a new user
const jane = await User.create({ firstName: 'Jane', lastName: 'Doe' });
console.log("Jane's auto-generated ID:", jane.id);
```

The `Model.create()` method is a shorthand for building an unsaved instance with `Model.build()` and saving the instance with `instance.save()`.

It is also possible to define which attributes can be set in the `create` method. This can be especially useful if you create database entries based on a form which can be filled by a user. Using that would, for example, allow you to restrict the `User` model to set only a username but not an admin flag (i.e., `isAdmin`):

```
const user = await User.create(
  {
    username: 'alice123',
    isAdmin: true,
  },
  { fields: ['username'] },
);
// Let's assume the default of isAdmin is false
```

```
console.log(user.username); // 'alice123'  
console.log(user.isAdmin); // false
```

Simple SELECT queries

You can read the whole table from the database with the `findAll` method:

```
// Find all users  
const users = await User.findAll();  
console.log(users.every(user => user instanceof User)); // true  
console.log('All users:', JSON.stringify(users, null, 2));
```

```
SELECT * FROM ...
```

Specifying attributes for SELECT queries

To select only some attributes, you can use the `attributes` option:

```
Model.findAll({  
  attributes: ['foo', 'bar'],  
});
```

```
SELECT foo, bar FROM ...
```

Attributes can be renamed using a nested array:

```
Model.findAll({  
  attributes: ['foo', ['bar', 'baz'], 'qux'],  
});
```

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

You can use `sequelize.fn` to do aggregations:

```
Model.findAll({
  attributes: ['foo', [sequelize.fn('COUNT', sequelize.col('hats')),
    'n_hats'], 'bar'],
});
```

```
SELECT foo, COUNT(hats) AS n_hats, bar FROM ...
```

When using aggregation function, you must give it an alias to be able to access it from the model. In the example above you can get the number of hats with `instance.n_hats`.

Sometimes it may be tiresome to list all the attributes of the model if you only want to add an aggregation:

```
// This is a tiresome way of getting the number of hats (along with every column)
Model.findAll({
  attributes: [
    'id',
    'foo',
    'bar',
    'baz',
    'qux',
    'hats', // We had to list all attributes...
    [sequelize.fn('COUNT', sequelize.col('hats')), 'n_hats'], // To add the aggregation...
  ],
});
```

```
// This is shorter, and less error prone because it still works if you add / remove attributes from your model later
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 ...
```

Similarly, it's also possible to remove a selected few attributes:

```
Model.findAll({
  attributes: { exclude: ['baz'] },
});
```

```
-- Assuming all columns are 'id', 'foo', 'bar', 'baz' and 'qux'
SELECT id, foo, bar, qux FROM ...
```

Applying WHERE clauses

The `where` option is used to filter the query. There are lots of operators to use for the `where` clause, available as Symbols from `Op`.

The basics

```
Post.findAll({
  where: {
    authorId: 2,
  },
});
// SELECT * FROM post WHERE authorId = 2;
```

Observe that no operator (from `Op`) was explicitly passed, so Sequelize assumed an equality comparison by default. The above code is equivalent to:

```
const { Op } = require('sequelize');
Post.findAll({
  where: {
    authorId: {
      [Op.eq]: 2,
    },
  },
});
// SELECT * FROM post WHERE authorId = 2;
```

Multiple checks can be passed:

```
Post.findAll({
  where: {
```

```

    authorId: 12,
    status: 'active',
  },
});
// SELECT * FROM post WHERE authorId = 12 AND status = 'active';

```

Just like Sequelize inferred the `Op.eq` operator in the first example, here Sequelize inferred that the caller wanted an `AND` for the two checks. The code above is equivalent to:

```

const { Op } = require('sequelize');
Post.findAll({
  where: {
    [Op.and]: [{ authorId: 12 }, { status: 'active' }],
  },
});
// SELECT * FROM post WHERE authorId = 12 AND status = 'active';

```

An `OR` can be easily performed in a similar way:

```

const { Op } = require('sequelize');
Post.findAll({
  where: {
    [Op.or]: [{ authorId: 12 }, { authorId: 13 }],
  },
});
// SELECT * FROM post WHERE authorId = 12 OR authorId = 13;

```

Since the above was an `OR` involving the same field, Sequelize allows you to use a slightly different structure which is more readable and generates the same behavior:

```

const { Op } = require('sequelize');
Post.destroy({
  where: {
    authorId: {
      [Op.or]: [12, 13],
    },
  },
});
// DELETE FROM post WHERE authorId = 12 OR authorId = 13;

```

Operators

Sequelize provides several operators.

```
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)
    [Op.match]: Sequelize.fn('to_tsquery', 'fat & rat') // match text
search for strings 'fat' and 'rat' (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
  }
}
});

```

Shorthand syntax for `Op.in`

Passing an array directly to the `where` option will implicitly use the `IN` operator:

```

Post.findAll({
  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);

```

Logical combinations with operators

The operators `Op.and`, `Op.or` and `Op.not` can be used to create arbitrarily complex nested logical comparisons.

Examples with `Op.and` and `Op.or`

```

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

Foo.findAll({
  where: {
    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%'
});

```

Examples with Op.not

```

Project.findAll({
  where: {
    name: 'Some Project',
    [Op.not]: [
      { id: [1, 2, 3] },
      {
        description: {
          [Op.like]: 'Hello%',
        },
      },
    ],
  },
});

```



```
    },  
  });  
};
```

The above will generate:

```
SELECT *  
FROM `Projects`  
WHERE (  
  `Projects`.`name` = 'Some Project'  
  AND NOT (  
    `Projects`.`id` IN (1,2,3)  
    AND  
    `Projects`.`description` LIKE 'Hello%'  
  )  
)
```

Advanced queries with functions (not just columns)

What if you wanted to obtain something like `WHERE char_length("content") = 7`?

```
Post.findAll({  
  where: sequelize.where(sequelize.fn('char_length',  
    sequelize.col('content')), 7),  
});  
// SELECT ... FROM "posts" AS "post" WHERE char_length("content") = 7
```

Note the usage of the `sequelize.fn` and `sequelize.col` methods, which should be used to specify an SQL function call and a table column, respectively. These methods should be used instead of passing a plain string (such as `char_length(content)`) because Sequelize needs to treat this situation differently (for example, using other symbol escaping approaches).

What if you need something even more complex?

```
Post.findAll({  
  where: {  
    [Op.or]: [  
      sequelize.where(sequelize.fn('char_length', sequelize.col('content')),  
6),  
      {  
        content: {  
          [Op.like]: 'Hello%',  
6}      }  
    ]  
  }  
});
```

```

    },
  },
  {
    [Op.and]: [
      { status: 'draft' },
      sequelize.where(sequelize.fn('char_length',
sequelize.col('content')), {
        [Op.gt]: 10,
      }),
    ],
  },
],
},
],
},
});

```

The above generates the following SQL:

```

SELECT
  ...
FROM "posts" AS "post"
WHERE (
  char_length("content") = 7
  OR
  "post"."content" LIKE 'Hello%'
  OR (
    "post"."status" = 'draft'
    AND
    char_length("content") > 10
  )
)

```

Postgres-only Range Operators

Range types can be queried with all supported operators.

Keep in mind, the provided range value can [define the bound inclusion/exclusion](#) as well.

```

[Op.contains]: 2,           // @> '2'::integer  (PG range contains element
operator)
[Op.contains]: [1, 2],      // @> [1, 2)      (PG range contains range
operator)
[Op.contained]: [1, 2],     // <@ [1, 2)      (PG range is contained by
operator)

```

<code>[Op.overlap]: [1, 2],</code>	<code>// && [1, 2)</code>	<i>(PG range overlap (have points in common) operator)</i>
<code>[Op.adjacent]: [1, 2],</code>	<code>// - - [1, 2)</code>	<i>(PG range is adjacent to operator)</i>
<code>[Op.strictLeft]: [1, 2],</code>	<code>// << [1, 2)</code>	<i>(PG range strictly left of operator)</i>
<code>[Op.strictRight]: [1, 2],</code>	<code>// >> [1, 2)</code>	<i>(PG range strictly right of operator)</i>
<code>[Op.noExtendRight]: [1, 2],</code>	<code>// &< [1, 2)</code>	<i>(PG range does not extend to the right of operator)</i>
<code>[Op.noExtendLeft]: [1, 2],</code>	<code>// &> [1, 2)</code>	<i>(PG range does not extend to the left of operator)</i>

Deprecated: Operator Aliases

In Sequelize v4, it was possible to specify strings to refer to operators, instead of using Symbols. This is now deprecated and heavily discouraged, and will probably be removed in the next major version. If you really need it, you can pass the `operatorAliases` option in the Sequelize constructor.

For example:

```
const { Sequelize, Op } = require('sequelize');
const sequelize = new Sequelize('sqlite::memory:', {
  operatorsAliases: {
    $gt: Op.gt,
  },
});

// Now we can use `$gt` instead of `[Op.gt]` in where clauses:
Foo.findAll({
  where: {
    $gt: 6, // Works like using [Op.gt]
  },
});
```

Simple UPDATE queries

Update queries also accept the `where` option, just like the read queries shown above.

```
// Change everyone without a Last name to "Doe"
await User.update(
  { lastName: 'Doe' },
  {
    where: {
      lastName: null,
    },
  },
);
```

Simple DELETE queries

Delete queries also accept the `where` option, just like the read queries shown above.

```
// Delete everyone named "Jane"
await User.destroy({
  where: {
    firstName: 'Jane',
  },
});
```

To destroy everything the `TRUNCATE` SQL can be used:

```
// Truncate the table
await User.destroy({
  truncate: true,
});
```

Creating in bulk

Sequelize provides the `Model.bulkCreate` method to allow creating multiple records at once, with only one query.

The usage of `Model.bulkCreate` is very similar to `Model.create`, by receiving an array of objects instead of a single object.

```
const captains = await Captain.bulkCreate([
  { name: 'Jack Sparrow' },
  { name: 'Davy Jones' }
]);
```

```
console.log(captains.length); // 2
console.log(captains[0] instanceof Captain); // true
console.log(captains[0].name); // 'Jack Sparrow'
console.log(captains[0].id); // 1 // (or another auto-generated value)
```

However, by default, `bulkCreate` does not run validations on each object that is going to be created (which `create` does). To make `bulkCreate` run these validations as well, you must pass the `validate: true` option. This will decrease performance. Usage example:

```
const Foo = sequelize.define('foo', {
  name: {
    type: DataTypes.TEXT,
    validate: {
      len: [4, 6],
    },
  },
});

// This will not throw an error, both instances will be created
await Foo.bulkCreate([{ name: 'abc123' }, { name: 'name too long' }]);

// This will throw an error, nothing will be created
await Foo.bulkCreate([{ name: 'abc123' }, { name: 'name too long' }], {
  validate: true,
});
```

If you are accepting values directly from the user, it might be beneficial to limit the columns that you want to actually insert. To support this, `bulkCreate()` accepts a `fields` option, an array defining which fields must be considered (the rest will be ignored).

```
await User.bulkCreate([{ username: 'foo' }, { username: 'bar', admin: true }], {
  fields: ['username'],
});
// Neither foo nor bar are admins.
```

Ordering and Grouping

Sequelize provides the `order` and `group` options to work with `ORDER BY` and `GROUP BY`.

Ordering

The `order` option takes an array of items to order the query by or a sequelize method. These *items* are themselves arrays in the form `[column, direction]`. The column will be escaped correctly and the direction will be checked in a whitelist of valid directions (such as `ASC`, `DESC`, `NULLS FIRST`, etc).

```
Subtask.findAll({
  order: [
    // Will escape title and validate DESC against a list of valid direction
    // parameters
    ['title', 'DESC'],

    // Will order by max(age)
    sequelize.fn('max', sequelize.col('age')),

    // Will order by max(age) DESC
    [sequelize.fn('max', sequelize.col('age')), 'DESC'],

    // Will order by otherfunction(`col1`, 12, 'lalala') DESC
    [sequelize.fn('otherfunction', sequelize.col('col1'), 12, 'lalala'),
    'DESC'],

    // Will order an associated model's createdAt using the model name as
    // the association's name.
    [Task, 'createdAt', 'DESC'],

    // Will order through an associated model's createdAt using the model
    // names as the associations' names.
    [Task, Project, 'createdAt', 'DESC'],

    // Will order by an associated model's createdAt using the name of the
    // association.
    ['Task', 'createdAt', 'DESC'],

    // Will order by a nested associated model's createdAt using the names
    // of the associations.
    ['Task', 'Project', 'createdAt', 'DESC'],

    // Will order by an associated model's createdAt using an association
    // object. (preferred method)
    [Subtask.associations.Task, 'createdAt', 'DESC'],

    // Will order by a nested associated model's createdAt using association
    // objects. (preferred method)
```

```

    [Subtask.associations.Task, Task.associations.Project, 'createdAt',
'DESC'],

    // Will order by an associated model's createdAt using a simple
association object.
    [{ model: Task, as: 'Task' }, 'createdAt', 'DESC'],

    // Will order by a nested associated model's createdAt simple
association objects.
    [{ model: Task, as: 'Task' }, { model: Project, as: 'Project' },
'createdAt', 'DESC'],
  ],

  // Will order by max age descending
  order: sequelize.literal('max(age) DESC'),

  // Will order by max age ascending assuming ascending is the default order
when direction is omitted
  order: sequelize.fn('max', sequelize.col('age')),

  // Will order by age ascending assuming ascending is the default order
when direction is omitted
  order: sequelize.col('age'),

  // Will order randomly based on the dialect (instead of fn('RAND') or
fn('RANDOM'))
  order: sequelize.random(),
});

Foo.findOne({
  order: [
    // will return `name`
    ['name'],
    // will return `username` DESC
    ['username', 'DESC'],
    // will return max(`age`)
    sequelize.fn('max', sequelize.col('age')),
    // will return max(`age`) DESC
    [sequelize.fn('max', sequelize.col('age')), 'DESC'],
    // will return otherfunction(`col1`, 12, 'laLaLa') DESC
    [sequelize.fn('otherfunction', sequelize.col('col1'), 12, 'laLaLa'),
'DESC'],
    // will return otherfunction(awesomefunction(`col`)) DESC, This nesting
is potentially infinite!
    [sequelize.fn('otherfunction', sequelize.fn('awesomefunction',
sequelize.col('col'))), 'DESC'],

```

```
  ],  
});
```

To recap, the elements of the order array can be the following:

- A string (which will be automatically quoted)
- An array, whose first element will be quoted, second will be appended verbatim
- An object with a `raw` field:
 - The content of `raw` will be added verbatim without quoting
 - Everything else is ignored, and if `raw` is not set, the query will fail
- A call to `Sequelize.fn` (which will generate a function call in SQL)
- A call to `Sequelize.col` (which will quote the column name)

Grouping

The syntax for grouping and ordering are equal, except that grouping does not accept a direction as last argument of the array (there is no `ASC`, `DESC`, `NULLS FIRST`, etc).

You can also pass a string directly to `group`, which will be included directly (verbatim) into the generated SQL. Use with caution and don't use with user generated content.

```
Project.findAll({ group: 'name' });  
// yields 'GROUP BY name'
```

Limits and Pagination

The `limit` and `offset` options allow you to work with limiting / pagination:

```
// Fetch 10 instances/rows  
Project.findAll({ limit: 10 });  
  
// Skip 8 instances/rows  
Project.findAll({ offset: 8 });  
  
// Skip 5 instances and fetch the 5 after that  
Project.findAll({ offset: 5, limit: 5 });
```

Usually these are used alongside the `order` option.

Utility methods

Sequelize also provides a few utility methods.

count

The `count` method simply counts the occurrences of elements in the database.

```
console.log(`There are ${await Project.count()} projects`);

const amount = await Project.count({
  where: {
    id: {
      [Op.gt]: 25,
    },
  },
});
console.log(`There are ${amount} projects with an id greater than 25`);
```

max, min and sum

Sequelize also provides the `max`, `min` and `sum` convenience methods.

Let's assume we have three users, whose ages are 10, 5, and 40.


```
await User.max('age'); // 40
await User.max('age', { where: { age: { [Op.lt]: 20 } } }); // 10
await User.min('age'); // 5
await User.min('age', { where: { age: { [Op.gt]: 5 } } }); // 10
await User.sum('age'); // 55
await User.sum('age', { where: { age: { [Op.gt]: 5 } } }); // 50
```

increment, decrement

Sequelize also provides the `increment` convenience method.

Let's assume we have a user, whose age is 10.

```
await User.increment({ age: 5 }, { where: { id: 1 } }); // Will increase age  
to 15  
await User.increment({ age: -5 }, { where: { id: 1 } }); // Will decrease  
age to 5
```

 [Edit this page](#)

Last updated on **Feb 28, 2025** by **renovate[bot]**