Querying

Attributes

To select only some attributes, you can use the `attributes` option. Most often, you pass an array:

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

SELECT foo, bar ...
```

Attributes can be renamed using a nested array:

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

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

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

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

SELECT COUNT(hats) AS no_hats ...
```

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.get('no_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...
Model.findAll({
  attributes: ['id', 'foo', 'bar', 'baz',
'quz', [sequelize.fn('COUNT',
sequelize.col('hats')), 'no_hats']]
});

// This is shorter, and less error prone
because it still works if you add / remove
```

```
  attributes: { include:
[[sequelize.fn('COUNT', sequelize.col('hats')),
'no_hats']] }
});

SELECT id, foo, bar, baz, quz, COUNT(hats) AS
no_hats ...
```

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

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

SELECT id, foo, bar, quz ...
```

Where

Whether you are querying with findAll/find or doing bulk updates/destroys you can pass a `where` object to filter the query.

`where` generally takes an object from attribute:value pairs, where value can be primitives for equality matches or keyed objects for other operators.

It's also possible to generate complex AND/OR conditions by nesting sets of `or` and `and` `Operators`.

Basics

```
const Op = Sequelize.Op;

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

Post.findAll({
  where: {
    authorId: 12,
    status: 'active'
  }
});
// SELECT * FROM post WHERE authorId = 12 AND
status = 'active';

Post.findAll({
  where: {
```

```
});
// SELECT * FROM post WHERE authorId = 12 OR
authorId = 13;

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

Post.destroy({
  where: {
    status: 'inactive'
  }
});
// DELETE FROM post WHERE status = 'inactive';

Post.update({
  updatedAt: null,
}, {
  where: {
    deletedAt: {
      [Op.ne]: null
    }
  }
});
// UPDATE post SET updatedAt = null WHERE
deletedAt NOT NULL;

Post.findAll({
  where:
sequelize.where(sequelize.fn('char_length',
sequelize.col('status')), 6)
});
// SELECT * FROM post WHERE char_length(status)
= 6;
```

Operators

Sequelize exposes symbol operators that can be used for to create more complex comparisons -

```
const Op = Sequelize.Op

[Op.and]: {a: 5}           // AND (a = 5)
[Op.or]: [{a: 5}, {a: 6}]  // (a = 5 OR a = 6)
[Op.gt]: 6,                // > 6
[Op.gte]: 6,               // >= 6
[Op.lt]: 10,               // < 10
[Op.lte]: 10,              // <= 10
[Op.ne]: 20,               // != 20
[Op.eq]: 3,                // = 3
[Op.not]: true,            // IS NOT TRUE
[Op.between]: [6, 10],     // BETWEEN 6 AND 10
[Op.notBetween]: [11, 15], // NOT BETWEEN 11
AND 15
```

```
[Op.like]: '%hat'         // LIKE '%hat'
[Op.notLike]: '%hat'      // NOT 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.like]: { [Op.any]: ['cat', 'hat']}
                          // LIKE ANY ARRAY['cat',
'hat'] - also works for iLike and notLike
[Op.overlap]: [1, 2]      // && [1, 2] (PG
array overlap operator)
[Op.contains]: [1, 2]     // @> [1, 2] (PG
array contains operator)
[Op.contained]: [1, 2]    // <@ [1, 2] (PG
array contained by operator)
[Op.any]: [2,3]           // ANY ARRAY[2,
3]::INTEGER (PG only)

[Op.col]: 'user.organization_id' // =
"user"."organization_id", with dialect specific
column identifiers, PG in this example
```

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.

```
// All the above equality and inequality
operators plus the following:

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

```
const Op = Sequelize.Op;

{
  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%'
```

Operators Aliases

Sequelize allows setting specific strings as aliases for operators -

```
const Op = Sequelize.Op;
const operatorsAliases = {
  $gt: Op.gt
}
const connection = new Sequelize(db, user,
pass, { operatorsAliases })

[Op.gt]: 6 // > 6
$gt: 6 // same as using Op.gt (> 6)
```

Operators security

Using Sequelize without any aliases improves security. Some frameworks automatically parse user input into js

Sequelize.

Not having any string aliases will make it extremely unlikely that operators could be injected but you should always properly validate and sanitize user input.

For backward compatibility reasons Sequelize sets the following aliases by default - $eq, $ne, $gte, $gt, $lte, $lt, $not, $in, $notIn, $is, $like, $notLike, $iLike, $notILike, $regexp, $notRegexp, $iRegexp, $notIRegexp, $between, $notBetween, $overlap, $contains, $contained, $adjacent, $strictLeft, $strictRight, $noExtendRight, $noExtendLeft, $and, $or, $any, $all, $values, $col

Currently the following legacy aliases are also set but are planned to be fully removed in the near future - ne, not, in, notIn, gte, gt, lte, lt, like, ilike, $ilike, nlike, $notlike, notilike, .., between, !.., notbetween, nbetween, overlap, &&, @>, <@

For better security it is highly advised to use `Sequelize.Op` and not depend on any string alias at all. You can limit alias your application will need by setting `operatorsAliases` option, remember to sanitize user input especially when you are directly passing them to Sequelize methods.

```
const Op = Sequelize.Op;

//use sequelize without any operators aliases
const connection = new Sequelize(db, user,
pass, { operatorsAliases: false });

//use sequelize with only alias for $and =>
Op.and
const connection2 = new Sequelize(db, user,
pass, { operatorsAliases: { $and: Op.and } });
```

Sequelize will warn you if you're using the default aliases and not limiting them if you want to keep using all default aliases (excluding legacy ones) without the warning you can pass the following operatorsAliases option -

```
const Op = Sequelize.Op;
const operatorsAliases = {
  $eq: Op.eq,
  $ne: Op.ne,
  $gte: Op.gte,
  $gt: Op.gt,
  $lte: Op.lte,
```

```
  $in: Op.in,
  $notIn: Op.notIn,
  $is: Op.is,
  $like: Op.like,
  $notLike: Op.notLike,
  $iLike: Op.iLike,
  $notILike: Op.notILike,
  $regexp: Op.regexp,
  $notRegexp: Op.notRegexp,
  $iRegexp: Op.iRegexp,
  $notIRegexp: Op.notIRegexp,
  $between: Op.between,
  $notBetween: Op.notBetween,
  $overlap: Op.overlap,
  $contains: Op.contains,
  $contained: Op.contained,
  $adjacent: Op.adjacent,
  $strictLeft: Op.strictLeft,
  $strictRight: Op.strictRight,
  $noExtendRight: Op.noExtendRight,
  $noExtendLeft: Op.noExtendLeft,
  $and: Op.and,
  $or: Op.or,
  $any: Op.any,
  $all: Op.all,
  $values: Op.values,
  $col: Op.col
};

const connection = new Sequelize(db, user,
pass, { operatorsAliases });
```

JSON

The JSON data type is supported by the PostgreSQL, SQLite and MySQL dialects only.

PostgreSQL

The JSON data type in PostgreSQL stores the value as plain text, as opposed to binary representation. If you simply want to store and retrieve a JSON representation, using JSON will take less disk space and less time to build from its input representation. However, if you want to do any operations on the JSON value, you should prefer the JSONB data type described below.

MSSQL

MSSQL does not have a JSON data type, however it does provide support for JSON stored as strings through certain functions since SQL Server 2016. Using these functions, you

returned values will need to be parsed separately.

```
// ISJSON - to test if a string contains valid
JSON
User.findAll({
  where: sequelize.where(sequelize.fn('ISJSON',
sequelize.col('userDetails')), 1)
})

// JSON_VALUE - extract a scalar value from a
JSON string
User.findAll({
  attributes: [[ sequelize.fn('JSON_VALUE',
sequelize.col('userDetails'),
'$.address.Line1'), 'address line 1']]
})

// JSON_VALUE - query a scalar value from a
JSON string
User.findAll({
  where:
sequelize.where(sequelize.fn('JSON_VALUE',
sequelize.col('userDetails'),
'$.address.Line1'), '14, Foo Street')
})

// JSON_QUERY - extract an object or array
User.findAll({
  attributes: [[ sequelize.fn('JSON_QUERY',
sequelize.col('userDetails'), '$.address'),
'full address']]
})
```

JSONB

JSONB can be queried in three different ways.

Nested object

```
{
  meta: {
    video: {
      url: {
        [Op.ne]: null
      }
    }
  }
}
```

Nested key

```
{
  "meta.audio.length": {
    [Op.gt]: 20
  }
}
```

```
{
  "meta": {
    [Op.contains]: {
      site: {
        url: 'http://google.com'
      }
    }
  }
}
```

Relations / Associations

```
// Find all projects with a least one task
where task.state === project.state
Project.findAll({
    include: [{
        model: Task,
        where: { state:
Sequelize.col('project.state') }
    }]
})
```

Pagination / Limiting

```
// 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 })
```

Ordering

`order` takes an array of items to order the query by or a sequelize method. Generally you will want to use a tuple/array of either attribute, direction or just direction to ensure proper escaping.

```
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'],
```

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

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

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

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

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

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

    // Will order by a nested associated
model's created_at using association objects.
(preferred method)
    [Subtask.associations.Task,
Task.associations.Project, 'createdAt',
'DESC'],

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

    // Will order by a nested associated
model's created_at 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'))
```

Table Hint

`tableHint` can be used to optionally pass a table hint when using mssql. The hint must be a value from `Sequelize.TableHints` and should only be used when absolutely necessary. Only a single table hint is currently supported per query.

Table hints override the default behavior of mssql query optimizer by specifing certain options. They only affect the table or view referenced in that clause.

```
const TableHints = Sequelize.TableHints;

Project.findAll({
  // adding the table hint NOLOCK
  tableHint: TableHints.NOLOCK
  // this will generate the SQL 'WITH (NOLOCK)'
})
```