Knex.js (1.0.3)

- » GitHub Repository
- » Support
- » FAQ
- » Change Log

Installation

- – Node.js
- – Browser
- **Config Options:**
- – client
- – debug
- – Getting parametrized instance
- – Async stacktraces
- – Pooling
- – afterCreate
- – acquireConnectionTimeout
- – fetchAsString
- – Migrations
- – postProcessResponse
- – wrapIdentifier
- – log

TypeScript SupportQuery Builder

- – **Identifier Syntax**
- – **constructor**
- – timeout
- – select
- – as
- – column
- – from
- – fromRaw
- – with
- – withRecursive
- – withMaterialized
- – withNotMaterialized
- – jsonExtract
- – jsonSet
- – jsonInsert
- – jsonRemove
- **Where Methods:**
- – where
- – whereNot
- – whereIn
- – whereNotIn
- – whereNull
- – whereNotNull
- – whereExists
- – whereNotExists
- – whereBetween
- – whereNotBetween
- – whereRaw
- – whereLike
- – whereILike
- – whereJsonObject
- – whereJsonPath
- – whereJsonSupersetOf
- – whereJsonSubsetOf
- **Join Methods:**
- – join
- – innerJoin
- – leftJoin
- – leftOuterJoin
- – rightJoin
- – rightOuterJoin
- – fullOuterJoin
- – crossJoin
- – joinRaw
- **On Methods:**
- – onIn
- – onNotIn
- – onNull
- – onNotNull
- – onExists
- – onNotExists
- – onBetween
- – onNotBetween
- – onJsonPathEquals
- **Clear Methods:**
- – clear
- – clearSelect (deprecated)
- – clearWhere (deprecated)
- – clearOrder (deprecated)
- – clearHaving (deprecated)

- – clearCounters
- **Having Methods:**
- – having
- – havingIn
- – havingNotIn
- – havingNull
- – havingNotNull
- – havingExists
- – havingNotExists
- – havingBetween
- – havingNotBetween
- – havingRaw
- – distinct
- – distinctOn
- – groupBy
- – groupByRaw
- – orderBy
- – orderByRaw
- – offset
- – limit
- – union
- – unionAll
- – insert
- – onConflict
- – ignore
- – merge
- – upsert
- – update
- – del / delete
- – using
- – returning
- – transacting
- – forUpdate
- – forShare
- – forNoKeyUpdate
- – forKeyShare
- – skipLocked
- – noWait
- – count
- – min
- – max
- – sum
- – avg
- – increment
- – decrement
- – hintComment
- – truncate
- – pluck
- – first
- – clone
- – rank
- – denseRank
- – rowNumber
- – partitionBy
- – modify
- – columnInfo
- – debug
- – connection
- – options
- - queryContext

Transactions

- – **overview**

Schema Builder

- – withSchema
- – createTable
- – createTableLike
- – renameTable
- – dropTable
- – hasColumn
- – hasTable
- – dropTableIfExists
- – table
- – alterTable
- – createView
- – createViewOrReplace
- – createMaterializedView
- – refreshMaterializedView
- – dropView
- – dropViewIfExists
- – dropMaterializedView
- – dropMaterializedViewIfExists
- – renameView
- – alterView

- – generateDdlCommands
- – raw
- – queryContext
- – dropSchema
- – dropSchemaIfExists
- **Schema Building:**
- – dropColumn
- – dropColumns
- – renameColumn
- – increments
- – integer
- – bigInteger
- – text
- – string
- – float
- – double
- – decimal
- – boolean
- – date
- – datetime
- – time
- – timestamp
- – timestamps
- – dropTimestamps
- – binary
- – enum / enu
- – json
- – jsonb
- – uuid
- – geometry
- – geography
- – point
- – comment
- – engine
- – charset
- – collate
- – inherits
- – specificType
- – index
- – dropIndex
- – setNullable
- – dropNullable
- – primary
- – unique
- – foreign
- – dropForeign
- – dropUnique
- – dropPrimary
- – queryContext
- **Chainable:**
- – alter
- – index
- – primary
- – unique
- – references
- – inTable
- – onDelete
- – onUpdate
- – defaultTo
- – unsigned
- – notNullable
- – nullable
- – first
- – after
- – comment
- – collate
- **View:**
- – columns
- – as
- – checkOption
- – localCheckOption
- – cascadedCheckOption
- **Check:**
- – check
- – checkPositive
- – checkNegative
- – checkIn
- – checkNotIn
- – checkBetween
- – checkLength
- – checkRegex
- – dropChecks

Raw

- – Raw Parameter Binding
- – Raw Expressions

- – Raw Queries
- - Wrapped Queries

Ref

- – Usage
- – withSchema
- – alias

Utility

- - Batch Insert
- - now
- - binToUuid
- - uuidToBin

Interfaces

- **Promises**
- – then
- – catch
- **Callbacks**
- – asCallback
- **Streams**
- – stream
- – pipe
- **Events**
- – query
- – query-error
- – query-response
- **Other:**
- – toString
- – toSQL
- – toSQL().toNative()

Migrations

- **CLI**
- – Migrations
- – Seed files
- – knexfile.js
- – esm
- **Migration API**
- – make
- – latest
- – rollback
  - up- down
- – currentVersion
- – list
- – unlock
- **Notes about locks**
- **Seed API**
- – make
- – run

SupportF.A.Q.Change Log

Show example query output as:
MySQL / MariaDB

Knex.js

Knex.js (pronounced /kəˈnɛks/) is a "batteries included" SQL query builder for **PostgreSQL**, **CockroachDB**, **MSSQL**, **MySQL**, **MariaDB**, **SQLite3**, **Better-SQLite3**, **Oracle**, and **Amazon Redshift** designed to be flexible, portable, and fun to use. It features both traditional node style callbacks as well as a promise interface for cleaner async flow control, a stream interface, full-featured query and schema builders, **transaction support (with savepoints)**, connection pooling and standardized responses between different query clients and dialects.

The project is hosted on GitHub, and has a comprehensive test suite.

Knex is available for use under the MIT software license.

You can report bugs and discuss features on the GitHub issues page, add pages to the wiki or send tweets to @kibertoad.

Thanks to all of the great contributions to the project.

Special thanks to Taylor Otwell and his work on the Laravel Query Builder, from which much of the builder's code and syntax was originally derived.

Latest Release: 1.0.3 - Change Log

Current Develop — CI passing

Installation

Knex can be used as an SQL query builder in both Node.JS and the browser, limited to WebSQL's constraints (like the inability to drop tables or read schemas). Composing SQL queries in the browser for execution on the server is highly discouraged, as this can be the cause of serious security vulnerabilities. The browser builds outside of WebSQL are primarily for learning purposes - for example, you can pop open the console and build queries on this page using the knex object.

Node.js

The primary target environment for Knex is Node.js, you will need to install the `knex` library, and then install the appropriate database library: `pg` for PostgreSQL, CockroachDB and Amazon Redshift, `pg-native` for PostgreSQL with native C++ `libpq` bindings (requires PostgresSQL installed to link against), `mysql` for MySQL or MariaDB, `sqlite3` for SQLite3, or `tedious` for MSSQL.

```
$ npm install knex --save

# Then add one of the following (adding a --save) flag:
$ npm install pg
$ npm install pg-native
$ npm install @vscode/sqlite3 # required for sqlite
$ npm install better-sqlite3
$ npm install mysql
$ npm install mysql2
$ npm install oracledb
$ npm install tedious
```

If you want to use CockroachDB or Redshift instance, you can use the `pg` driver.

If you want to use a MariaDB instance, you can use the `mysql` driver.

Browser

Knex can be built using a JavaScript build tool such as browserify or webpack. In fact, this documentation uses a webpack build which includes knex. View source on this page to see the browser build in-action (the global `knex` variable).

Initializing the Library

The `knex` module is itself a function which takes a configuration object for Knex, accepting a few parameters. The `client` parameter is required and determines which client adapter will be used with the library.

```
const knex = require('knex')({
  client: 'mysql',
  connection: {
    host : '127.0.0.1',
    port : 3306,
    user : 'your_database_user',
    password : 'your_database_password',
    database : 'myapp_test'
  }
});
```

The connection options are passed directly to the appropriate database client to create the connection, and may be either an object, a connection string, or a function returning an object:

Note: Knex's PostgreSQL client allows you to set the initial search path for each connection automatically using an additional option "searchPath" as shown below.

```
const pg = require('knex')({
  client: 'pg',
  connection: process.env.PG_CONNECTION_STRING,
  searchPath: ['knex', 'public'],
});
```

Note: When you use the SQLite3 or Better-SQLite3 adapter, there is a filename required, not a network connection. For example:

```
const knex = require('knex')({
  client: 'sqlite3', // or 'better-sqlite3'
  connection: {
    filename: "./mydb.sqlite"
  }
});
```

Note: You can also run either SQLite3 or Better-SQLite3 with an in-memory database by providing `:memory:` as the filename. For example:

```
const knex = require('knex')({
  client: 'sqlite3', // or 'better-sqlite3'
  connection: {
    filename: ":memory:"
  }
});
```

Note: When you use the SQLite3 adapter, you can set flags used to open the connection. For example:

```
const knex = require('knex')({
  client: 'sqlite3',
  connection: {
    filename: "file:memDb1?mode=memory&cache=shared",
    flags: ['OPEN_URI', 'OPEN_SHAREDCACHE']
  }
});
```

Note: The database version can be added in knex configuration, when you use the PostgreSQL adapter to connect a non-standard database.

```
const knex = require('knex')({
  client: 'pg',
  version: '7.2',
  connection: {
    host : '127.0.0.1',
    port : 3306,
    user : 'your_database_user',
    password : 'your_database_password',
    database : 'myapp_test'
  }
});
```

```
const knex = require('knex')({
  client: 'mysql',
  version: '5.7',
  connection: {
    host : '127.0.0.1',
    port : 3306,
    user : 'your_database_user',
    password : 'your_database_password',
    database : 'myapp_test'
  }
});
```

A function can be used to determine the connection configuration dynamically. This function receives no parameters, and returns either a configuration object or a promise for a configuration object.

```
const knex = require('knex')({
  client: 'sqlite3',
  connection: () => ({
    filename: process.env.SQLITE_FILENAME
  })
});
```

By default, the configuration object received via a function is cached and reused for all connections. To change this behavior, an `expirationChecker` function can be returned as part of the configuration object. The `expirationChecker` is consulted before trying to create new connections, and in case it returns `true`, a new configuration object is retrieved. For example, to work with an authentication token that has a limited lifespan:

```
const knex = require('knex')({
  client: 'postgres',
  connection: async () => {
    const { token, tokenExpiration } = await someCallToGetTheToken();
    return {
      host : 'your_host',
      port : 3306,
      user : 'your_database_user',
      password : token,
      database : 'myapp_test',
      expirationChecker: () => {
        return tokenExpiration <= Date.now();
      }
    };
  }
});
```

You can also connect via an unix domain socket, which will ignore host and port.

```
const knex = require('knex')({
  client: 'mysql',
  connection: {
    socketPath : '/path/to/socket.sock',
    user : 'your_database_user',
    password : 'your_database_password',
    database : 'myapp_test'
  }
});
```

`userParams` is an optional parameter that allows you to pass arbitrary parameters which will be accessible via `knex.userParams` property:

```
const knex = require('knex')({
  client: 'mysql',
  connection: {
    host : '127.0.0.1',
    port : 3306,
    user : 'your_database_user',
    password : 'your_database_password',
    database : 'myapp_test'
  },
  userParams: {
    userParam1: '451'
  }
});
```

Initializing the library should normally only ever happen once in your application, as it creates a connection pool for the current database, you should use the instance returned from the initialize call throughout your library.

Specify the client for the particular flavour of SQL you are interested in.

```
const pg = require('knex')({client: 'pg'});
knex('table').insert({a: 'b'}).returning('*').toString();
// "insert into "table" ("a") values ('b')"

pg('table').insert({a: 'b'}).returning('*').toString();
// "insert into "table" ("a") values ('b') returning *"
```

Getting parametrized instance

You can call method `withUserParams` on a Knex instance if you want to get a copy (with same connections) with custom parameters (e. g. to execute same migrations with different parameters)

```
const knex = require('knex')({
  // Params
});

const knexWithParams = knex.withUserParams({customUserParam: 'table1'});
const customUserParam = knexWithParams.userParams.customUserParam;
```

Debugging

Passing a `debug: true` flag on your initialization object will turn on debugging for all queries.

asyncStackTraces

Passing an `asyncStackTraces: true` flag on your initialization object will turn on stack trace capture for all query builders, raw queries and schema builders. When a DB driver returns an error, this previously captured stack trace is thrown instead of a new one. This helps to mitigate default behaviour of `await` in node.js/V8 which blows the stack away. This has small performance overhead, so it is advised to use only for development. Turned off by default.

Pooling

The client created by the configuration initializes a connection pool, using the tarn.js library. This connection pool has a default setting of a `min: 2, max: 10` for the MySQL and PG libraries, and a single connection for sqlite3 (due to issues with utilizing multiple connections on a single file). To change the config settings for the pool, pass a `pool` option as one of the keys in the initialize block.

Checkout the tarn.js library for more information.

```
const knex = require('knex')({
  client: 'mysql',
  connection: {
    host : '127.0.0.1',
    port : 3306,
    user : 'your_database_user',
    password : 'your_database_password',
    database : 'myapp_test'
  },
  pool: { min: 0, max: 7 }
});
```

If you ever need to explicitly teardown the connection pool, you may use `knex.destroy([callback])`. You may use `knex.destroy` by passing a callback, or by chaining as a promise, just not both. To manually initialize a destroyed connection pool, you may use knex.initialize([config]), if no config is passed, it will use the first knex configuration used.

afterCreate

`afterCreate` callback (rawDriverConnection, done) is called when the pool aquires a new connection from the database server. done(err, connection) callback must be called for `knex` to be able to decide if the connection is ok or if it should be discarded right away from the pool.

```
const knex = require('knex')({
  client: 'pg',
  connection: {...},
  pool: {
    afterCreate: function (conn, done) {
      // in this example we use pg driver's connection API
      conn.query('SET timezone="UTC";', function (err) {
        if (err) {
          // first query failed, return error and don't try to make next query
          done(err, conn);
        } else {
          // do the second query...
          conn.query('SELECT set_limit(0.01);', function (err) {
            // if err is not falsy, connection is discarded from pool
            // if connection aquire was triggered by a query the error is passed to query promise
            done(err, conn);
          });
        }
      });
    }
  }
});
```

acquireConnectionTimeout

`acquireConnectionTimeout` defaults to 60000ms and is used to determine how long knex should wait before throwing a timeout error when acquiring a connection is not possible. The most common cause for this is using up all the pool for transaction connections and then attempting to run queries outside of transactions while the pool is still full. The error thrown will provide information on the query the connection was for to simplify the job of locating the culprit.

```
const knex = require('knex')({
  client: 'pg',
  connection: {...},
  pool: {...},
  acquireConnectionTimeout: 10000
});
```

fetchAsString

Utilized by Oracledb. An array of types. The valid types are 'DATE', 'NUMBER' and 'CLOB'. When any column having one of the specified types is queried, the column data is returned as a string instead of the default representation.

```
const knex = require('knex')({
  client: 'oracledb',
  connection: {...},
  fetchAsString: [ 'number', 'clob' ]
});
```

Migrations

For convenience, the any migration configuration may be specified when initializing the library. Read the Migrations section for more information and a full list of configuration options.

```
const knex = require('knex')({
  client: 'mysql',
  connection: {
    host : '127.0.0.1',
    port : 3306,
    user : 'your_database_user',
    password : 'your_database_password',
```

```
    database : 'myapp_test'
  },
  migrations: {
    tableName: 'migrations'
  }
});
```

postProcessResponse

Hook for modifying returned rows, before passing them forward to user. One can do for example snake_case -> camelCase conversion for returned columns with this hook. The `queryContext` is only available if configured for a query builder instance via queryContext.

```
const knex = require('knex')({
  client: 'mysql',
  // overly simplified snake_case -> camelCase converter
  postProcessResponse: (result, queryContext) => {
    // TODO: add special case for raw results (depends on dialect)
    if (Array.isArray(result)) {
      return result.map(row => convertToCamel(row));
    } else {
      return convertToCamel(result);
    }
  }
});
```

wrapIdentifier

Knex supports transforming identifier names automatically to quoted versions for each dialect. For example `'Table.columnName as foo'` for PostgreSQL is converted to "Table"."columnName" as "foo".

With `wrapIdentifier` one may override the way how identifiers are transformed. It can be used to override default functionality and for example to help doing `camelCase` -> `snake_case` conversion.

Conversion function `wrapIdentifier(value, dialectImpl, context): string` gets each part of the identifier as a single `value`, the original conversion function from the dialect implementation and the `queryContext`, which is only available if configured for a query builder instance via builder.queryContext, and for schema builder instances via schema.queryContext or table.queryContext. For example, with the query builder, `knex('table').withSchema('foo').select('table.field as otherName').where('id', 1)` will call `wrapIdentifier` converter for following values `'table'`, `'foo'`, `'table'`, `'field'`, `'otherName'` and `'id'`.

```
const knex = require('knex')({
  client: 'mysql',
  // overly simplified camelCase -> snake_case converter
  wrapIdentifier: (value, origImpl, queryContext) => origImpl(convertToSnakeCase(value))
});
```

log

Knex contains some internal log functions for printing warnings, errors, deprecations, and debug information when applicable. These log functions typically log to the console, but can be overwritten using the log option and providing alternative functions. Different log functions can be used for separate knex instances.

```
const knex = require('knex')({
   log: {
    warn(message) {
    },
    error(message) {
    },
    deprecate(message) {
    },
    debug(message) {
    },
  }
});
```

TypeScript Support

While knex is written in JavaScript, officially supported TypeScript bindings are available (within the knex npm package).

However it is to be noted that TypeScript support is currently best-effort. Knex has a very flexible API and not all usage patterns can be type-checked and in most such cases we err on the side of flexibility. In particular, lack of type errors doesn't currently guarantee that the generated queries will be correct and therefore writing tests for them is recommended even if you are using TypeScript.

Many of the APIs accept `TRecord` and `TResult` type parameters, using which we can specify the type of a row in the database table and the type of the result of the query respectively. This is helpful for auto-completion when using TypeScript-aware editors like VSCode.

To reduce boilerplate and add inferred types, you can augment `Tables` interface in `'knex/types/tables'` module.

```
import { knex } from 'knex';

declare module 'knex/types/tables' {
  interface User {
    id: number;
    name: string;
    created_at: string;
    updated_at: string;
  }

  interface Tables {
    // This is same as specifying `knex<User>('users')`
    users: User;
    // For more advanced types, you can specify separate type
    // for base model, "insert" type and "update" type.
    // But first: notice that if you choose to use this,
    // the basic typing showed above can be ignored.
    // So, this is like specifying
    //    knex
    //    .insert<{ name: string }>({ name: 'name' })
```

```
    //    .into<{ name: string, id: number }>('users')
    users_composite: Knex.CompositeTableType<
      // This interface will be used for return type and
      // `where`, `having` etc where full type is required
      User,
      // Specifying "insert" type will also make sure
      // data matches interface in full. Meaning
      // if interface is `{ a: string, b: string }`,
      // `insert({ a: '' })` will complain about missing fields.
      //
      // For example, this will require only "name" field when inserting
      // and make created_at and updated_at optional.
      // And "id" can't be provided at all.
      // Defaults to "base" type.
      Pick<User, 'name'> & Partial<Pick<User, 'created_at' | 'updated_at'>>,
      // This interface is used for "update()" calls.
      // As opposed to regular specifying interface only once,
      // when specifying separate update interface, user will be
      // required to match it  exactly. So it's recommended to
      // provide partial interfaces for "update". Unless you want to always
      // require some field (e.g., `Partial<User> & { updated_at: string }`
      // will allow updating any field for User but require updated_at to be
      // always provided as well.
      //
      // For example, this wil allow updating all fields except "id".
      // "id" will still be usable for `where` clauses so
      //      knex('users_composite')
      //      .update({ name: 'name2' })
      //      .where('id', 10)`
      // will still work.
      // Defaults to Partial "insert" type
      Partial<Omit<User, 'id'>>
    >;
  }
}
```

Knex Query Builder

The heart of the library, the knex query builder is the interface used for building and executing standard SQL queries, such as `select`, `insert`, `update`, `delete`.

Identifier Syntax

In many places in APIs identifiers like table name or column name can be passed to methods.

Most commonly one needs just plain `tableName.columnName`, `tableName` or `columnName`, but in many cases one also needs to pass an alias how that identifier is referred later on in the query.

There are two ways to declare an alias for identifier. One can directly give `as aliasName` suffix for the identifier (e.g. `identifierName as aliasName`) or one can pass an object `{ aliasName: 'identifierName' }`.

If the object has multiple aliases `{ alias1: 'identifier1', alias2: 'identifier2' }`, then all the aliased identifiers are expanded to comma separated list.

NOTE: identifier syntax has no place for selecting schema, so if you are doing `schemaName.tableName`, query might be rendered wrong. Use `.withSchema('schemaName')` instead.

```
knex({ a: 'table', b: 'table' })
  .select({
    aTitle: 'a.title',
    bTitle: 'b.title'
  })
  .whereRaw('?? = ??', ['a.column_1', 'b.column_2'])
Outputs:
select `a`.`title` as `aTitle`, `b`.`title` as `bTitle` from `table` as `a`, `table` as `b` where `a`.`column_1` = `b`.`column_2`
```

knex — `knex(tableName, options={only: boolean}) / knex.[methodName]`

The query builder starts off either by specifying a tableName you wish to query against, or by calling any method directly on the knex object. This kicks off a jQuery-like chain, with which you can call additional query builder methods as needed to construct the query, eventually calling any of the interface methods, to either convert toString, or execute the query with a promise, callback, or stream. Optional second argument for passing options:* **only**: if `true`, the ONLY keyword is used before the `tableName` to discard inheriting tables' data. **NOTE:** only supported in PostgreSQL for now.

Usage with TypeScript

If using TypeScript, you can pass the type of database row as a type parameter to get better autocompletion support down the chain.

```
interface User {
  id: number;
  name: string;
  age: number;
}

knex('users')
  .where('id')
  .first(); // Resolves to any

knex<User>('users') // User is the type of row in database
  .where('id', 1) // Your IDE will be able to help with the completion of id
  .first(); // Resolves to User | undefined
```

It is also possible to take advantage of auto-completion support (in TypeScript-aware IDEs) with generic type params when writing code in plain JavaScript through JSDoc comments.

```
/**
 * @typedef {Object} User
 * @property {number} id
 * @property {number} age
 * @property {string} name
```

```
 *
 * @returns {Knex.QueryBuilder<User, {}>}
 */
const Users = () => knex('Users')

Users().where('id', 1) // 'id' property can be autocompleted by editor
```

Caveat with type inference and mutable fluent APIs

Most of the knex APIs mutate current object and return it. This pattern does not work well with type-inference.

```
knex<User>('users')
  .select('id')
  .then((users) => { // Type of users is inferred as Pick<User, "id">[]
    // Do something with users
  });

knex<User>('users')
  .select('id')
  .select('age')
  .then((users) => { // Type of users is inferred as Pick<User, "id" | "age">[]
    // Do something with users
  });

// The type of usersQueryBuilder is determined here
const usersQueryBuilder = knex<User>('users').select('id');

if (someCondition) {
  // This select will not change the type of usersQueryBuilder
  // We can not change the type of a pre-declared variable in TypeScript
  usersQueryBuilder.select('age');
}
usersQueryBuilder.then((users) => {
  // Type of users here will be Pick<User, "id">[]
  // which may not be what you expect.
});

// You can specify the type of result explicitly through a second type parameter:
const queryBuilder = knex<User, Pick<User, "id" | "age">>('users');

// But there is no type constraint to ensure that these properties have actually been
// selected.

// So, this will compile:
queryBuilder.select('name').then((users) => {
  // Type of users is Pick<User, "id"> but it will only have name
})
```

If you don't want to manually specify the result type, it is recommended to always use the type of last value of the chain and assign result of any future chain continuation to a separate variable (which will have a different type).

timeout — `.timeout(ms, options={cancel: boolean})`

Sets a timeout for the query and will throw a TimeoutError if the timeout is exceeded. The error contains information about the query, bindings, and the timeout that was set. Useful for complex queries that you want to make sure are not taking too long to execute. Optional second argument for passing options:* **cancel**: if `true`, cancel query if timeout is reached. **NOTE:** only supported in MySQL and PostgreSQL for now.

```
knex.select().from('books').timeout(1000)
Outputs:
select * from `books`

knex.select().from('books').timeout(1000, {cancel: true}) // MySQL and PostgreSQL only
Outputs:
select * from `books`
```

select — `.select([*columns])`

Creates a select query, taking an optional array of columns for the query, eventually defaulting to * if none are specified when the query is built. The response of a select call will resolve with an array of objects selected from the database.

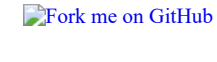
```
knex.select('title', 'author', 'year').from('books')
Outputs:
select `title`, `author`, `year` from `books`

knex.select().table('books')
Outputs:
select * from `books`
```

Usage with TypeScript

We are generally able to infer the result type based on the columns being selected as long as the select arguments match exactly the key names in record type. However, aliasing and scoping can get in the way of inference.

```
knex.select('id').from<User>('users'); // Resolves to Pick<User, "id">[]

knex.select('users.id').from<User>('users'); // Resolves to any[]
// ^ TypeScript doesn't provide us a way to look into a string and infer the type
//   from a substring, so we fall back to any

// We can side-step this using knex.ref:
knex.select(knex.ref('id').withSchema('users')).from<User>('users'); // Resolves to Pick<User, "id">[]

knex.select('id as identifier').from<User>('users'); // Resolves to any[], for same reason as above

// Refs are handy here too:
knex.select(knex.ref('id').as('identifier')).from<User>('users'); // Resolves to { identifier: number; }[]
```

as — `.as(name)`

Allows for aliasing a subquery, taking the string you wish to name the current query. If the query is not a sub-query, it will be ignored. 

```
knex.avg('sum_column1').from(function() {
  this.sum('column1 as sum_column1').from('t1').groupBy('column1').as('t1')
}).as('ignored_alias')
Outputs:
select avg(`sum_column1`) from (select sum(`column1`) as `sum_column1` from `t1` group by `column1`) as `t1`
```

column — `.column(columns)`

Specifically set the columns to be selected on a select query, taking an array, an object or a list of column names. Passing an object will automatically alias the columns with the given keys.

```
knex.column('title', 'author', 'year').select().from('books')
Outputs:
select `title`, `author`, `year` from `books`

knex.column(['title', 'author', 'year']).select().from('books')
Outputs:
select `title`, `author`, `year` from `books`

knex.column('title', {by: 'author'}, 'year').select().from('books')
Outputs:
select `title`, `author` as `by`, `year` from `books`
```

from — `.from([tableName], options={only: boolean})`

Specifies the table used in the current query, replacing the current table name if one has already been specified. This is typically used in the sub-queries performed in the advanced where or union methods. Optional second argument for passing options:* **only**: if `true`, the ONLY keyword is used before the `tableName` to discard inheriting tables' data. **NOTE:** only supported in PostgreSQL for now.

```
knex.select('*').from('users')
Outputs:
select * from `users`
```

Usage with TypeScript

We can specify the type of database row through the TRecord type parameter

```
knex.select('id').from('users'); // Resolves to any[]

knex.select('id').from<User>('users'); // Results to Pick<User, "id">[]
```

fromRaw — `.fromRaw(sql, [bindings])`

```
knex.select('*').fromRaw('(select * from "users" where "age" > ?)', '18')
Outputs:
select * from (select * from "users" where "age" > '18')
```

with — `.with(alias, [columns], callback|builder|raw)`

Add a "with" clause to the query. "With" clauses are supported by PostgreSQL, Oracle, SQLite3 and MSSQL. An optional column list can be provided after the alias; if provided, it must include at least one column name.

```
knex.with('with_alias', knex.raw('select * from "books" where "author" = ?', 'Test')).select('*').from('with_alias')
Outputs:
with `with_alias` as (select * from "books" where "author" = 'Test') select * from `with_alias`

knex.with('with_alias', ["title"], knex.raw('select "title" from "books" where "author" = ?', 'Test')).select('*').from('with_alias')
Outputs:
with `with_alias`(`title`) as (select "title" from "books" where "author" = 'Test') select * from `with_alias`

knex.with('with_alias', (qb) => {
  qb.select('*').from('books').where('author', 'Test')
}).select('*').from('with_alias')
Outputs:
with `with_alias` as (select * from `books` where `author` = 'Test') select * from `with_alias`
```

withRecursive — `.withRecursive(alias, [columns], callback|builder|raw)`

Identical to the `with` method except "recursive" is appended to "with" (or not, as required by the target database) to make self-referential CTEs possible. Note that some databases, such as Oracle, require a column list be provided when using an rCTE.

```
knex.withRecursive('ancestors', (qb) => {
  qb.select('*').from('people').where('people.id', 1).union((qb) => {
    qb.select('*').from('people').join('ancestors', 'ancestors.parentId', 'people.id')
  })
}).select('*').from('ancestors')
Outputs:
with recursive `ancestors` as (select * from `people` where `people`.`id` = 1 union select * from `people` inner join `ancestors` on `ancestors`.`pare

knex.withRecursive('family', ['name', 'parentName'], (qb) => {
  qb.select('name', 'parentName')
    .from('folks')
    .where({ name: 'grandchild' })
    .unionAll((qb) =>
      qb
        .select('folks.name', 'folks.parentName')
        .from('folks')
        .join(
          'family',
          knex.ref('family.parentName'),
          knex.ref('folks.name')
        )
    )
})
.select('name')
.from('family')
Outputs:
with recursive `family`(`name`, `parentName`) as (select `name`, `parentName` from `folks` where `name` = 'grandchild' union all select `folks`.`name`
```

withMaterialized — `.withMaterialized(alias, [columns], callback|builder|raw)`

Add a "with" materialized clause to the query. "With" materialized clauses are supported by PostgreSQL and SQLite3. An optional column list can be provided after the alias; if provided, it must include at least one column name.

```
knex.withMaterialized('with_alias', knex.raw('select * from "books" where "author" = ?', 'Test')).select('*').from('with_alias')
Error:
With materialized is not supported by this dialect
```

```
knex.withMaterialized('with_alias', ["title"], knex.raw('select "title" from "books" where "author" = ?', 'Test')).select('*').from('with_alias')
Error:
With materialized is not supported by this dialect
```

```
knex.withMaterialized('with_alias', (qb) => {
  qb.select('*').from('books').where('author', 'Test')
}).select('*').from('with_alias')
Error:
With materialized is not supported by this dialect
```

withNotMaterialized — `.withNotMaterialized(alias, [columns], callback|builder|raw)`

Add a "with" not materialized clause to the query. "With" not materialized clauses are supported by PostgreSQL and SQLite3. An optional column list can be provided after the alias; if provided, it must include at least one column name.

```
knex.withNotMaterialized('with_alias', knex.raw('select * from "books" where "author" = ?', 'Test')).select('*').from('with_alias')
Error:
With materialized is not supported by this dialect
```

```
knex.withNotMaterialized('with_alias', ["title"], knex.raw('select "title" from "books" where "author" = ?', 'Test')).select('*').from('with_alias')
Error:
With materialized is not supported by this dialect
```

```
knex.withNotMaterialized('with_alias', (qb) => {
  qb.select('*').from('books').where('author', 'Test')
}).select('*').from('with_alias')
Error:
With materialized is not supported by this dialect
```

withSchema — `.withSchema([schemaName])`

Specifies the schema to be used as prefix of table name.

```
knex.withSchema('public').select('*').from('users')
Outputs:
select * from `public`.`users`
```

jsonExtract — `.jsonExtract(column|builder|raw|array[], path, [alias], [singleValue])`

Extract a value from a json column given a JsonPath. An alias can be specified. The singleValue booleancan be used to specify, with Oracle or MSSQL, if the value returned by the function is a single value or an array/object value.An array of arrays can be used to specify multiple extractions with one call to this function.

```
knex('accounts').jsonExtract('json_col', '$.name')
Outputs:
select json_unquote(json_extract(`json_col`, '$.name')) from `accounts`
```

```
knex('accounts').jsonExtract('json_col', '$.name', 'accountName')
Outputs:
select json_unquote(json_extract(`json_col`, '$.name')) as `accountName` from `accounts`
```

```
knex('accounts').jsonExtract('json_col', '$.name', 'accountName', true)
Outputs:
select json_unquote(json_extract(`json_col`, '$.name')) as `accountName` from `accounts`
```

```
knex('accounts').jsonExtract([
  ['json_col', '$.name', 'accountName'],
  ['json_col', '$.lastName', 'accountLastName']
])
Outputs:
select json_unquote(json_extract(`json_col`, '$.name')) as `accountName`, json_unquote(json_extract(`json_col`, '$.lastName')) as `accountLastName` fr
```

All json*() functions can be used directly from knex object and can be nested.

```
knex('cities').jsonExtract([
      [knex.jsonRemove('population', '$.min'), '$', 'withoutMin'],
      [knex.jsonRemove('population', '$.max'), '$', 'withoutMax'],
      [
        knex.jsonSet('population', '$.current', '1234'),
        '$',
        'currentModified',
      ]
    ])
Outputs:
select json_unquote(json_extract(json_remove(`population`,'$.min'), '$')) as `withoutMin`, json_unquote(json_extract(json_remove(`population`,'$.max'
```

jsonSet — `.jsonSet(column|builder|raw, path, value, [alias])`

Return a json value/object/array where a given value is set at the given JsonPath. Value can be single value or json object. If a value already exists at the given place, the value is replaced.Not supported by Redshift and versions before Oracle 21c.

```
knex('accounts').jsonSet('json_col', '$.name', 'newName', 'newNameCol')
Outputs:
select json_set(`json_col`, '$.name', 'newName') as `newNameCol` from `accounts`
```

```
knex('accounts').jsonSet('json_col', '$.name', { "name": "newName" }, 'newNameCol')
Outputs:
select json_set(`json_col`, '$.name', {"name":"newName"}) as `newNameCol` from `accounts`
```

jsonInsert — `.jsonInsert(column|builder|raw, path, value, [alias])`

Return a json value/object/array where a given value is inserted at the given JsonPath. Value can be single value or json object. If a value exists at the given path, the value is not replaced.Not supported by Redshift and versions before Oracle 21c.

```
knex('accounts').jsonInsert('json_col', '$.name', 'newName', 'newNameCol')
Outputs:
select json_insert(`json_col`, '$.name', 'newName') as `newNameCol` from `accounts`

knex('accounts').jsonInsert('json_col', '$.name', { "name": "newName" }, 'newNameCol')
Outputs:
select json_insert(`json_col`, '$.name', {"name":"newName"}) as `newNameCol` from `accounts`

knex('accounts').jsonInsert(knex.jsonExtract('json_col', '$.otherAccount'), '$.name', { "name": "newName" }, 'newNameCol')
Outputs:
select json_insert(json_unquote(json_extract(`json_col`, '$.otherAccount')), '$.name', {"name":"newName"}) as `newNameCol` from `accounts`
```

jsonRemove — `.jsonRemove(column|builder|raw, path, [alias])`

Return a json value/object/array where a given value is removed at the given JsonPath.Not supported by Redshift and versions before Oracle 21c.

```
knex('accounts').jsonRemove('json_col', '$.name', 'colWithRemove')
Outputs:
select json_remove(`json_col`,'$.name') as `colWithRemove` from `accounts`

knex('accounts').jsonInsert('json_col', '$.name', { "name": "newName" }, 'newNameCol')
Outputs:
select json_insert(`json_col`, '$.name', {"name":"newName"}) as `newNameCol` from `accounts`
```

Where Clauses

Several methods exist to assist in dynamic where clauses. In many places functions may be used in place of values, constructing subqueries. In most places existing knex queries may be used to compose sub-queries, etc. Take a look at a few of the examples for each method for instruction on use:

Important: Supplying knex with an `undefined` value to any of the `where` functions will cause knex to throw an error during sql compilation. This is both for yours and our sake. Knex cannot know what to do with undefined values in a where clause, and generally it would be a programmatic error to supply one to begin with. The error will throw a message containing the type of query and the compiled query-string. Example:

```
knex('accounts')
  .where('login', undefined)
  .select()
  .toSQL()
Error:
Undefined binding(s) detected when compiling SELECT. Undefined column(s): [login] query: select * from `accounts` where `login` = ?
```

where — `.where(~mixed~) / .orWhere`

Object Syntax:

```
knex('users').where({
  first_name: 'Test',
  last_name:  'User'
}).select('id')
Outputs:
select `id` from `users` where `first_name` = 'Test' and `last_name` = 'User'
```

Key, Value:

```
knex('users').where('id', 1)
Outputs:
select * from `users` where `id` = 1
```

Functions:

```
knex('users')
.where((builder) =>
  builder.whereIn('id', [1, 11, 15]).whereNotIn('id', [17, 19])
)
.andWhere(function() {
  this.where('id', '>', 10)
})
Outputs:
select * from `users` where (`id` in (1, 11, 15) and `id` not in (17, 19)) and (`id` > 10)
```

Grouped Chain:

```
knex('users').where(function() {
  this.where('id', 1).orWhere('id', '>', 10)
}).orWhere({name: 'Tester'})
Outputs:
select * from `users` where (`id` = 1 or `id` > 10) or (`name` = 'Tester')
```

Operator:

```
knex('users').where('columnName', 'like', '%rowlikeme%')
Outputs:
select * from `users` where `columnName` like '%rowlikeme%'
```

The above query demonstrates the common use case of returning all users for which a specific pattern appears within a designated column.

```
knex('users').where('votes', '>', 100)
Outputs:
select * from `users` where `votes` > 100

const subquery = knex('users').where('votes', '>', 100).andWhere('status', 'active').orWhere('name', 'John').select('id');

knex('accounts').where('id', 'in', subquery)
Outputs:
select * from `accounts` where `id` in (select `id` from `users` where `votes` > 100 and `status` = 'active' or `name` = 'John')
```

.orWhere with an object automatically wraps the statement and creates an `or (and - and - and)` clause

```
knex('users').where('id', 1).orWhere({votes: 100, user: 'knex'})
Outputs:
select * from `users` where `id` = 1 or (`votes` = 100 and `user` = 'knex')
```

whereNot — .whereNot(~mixed~) / .orWhereNot

Object Syntax:

```
knex('users').whereNot({
  first_name: 'Test',
  last_name:  'User'
}).select('id')
Outputs:
select `id` from `users` where not `first_name` = 'Test' and not `last_name` = 'User'
```

Key, Value:

```
knex('users').whereNot('id', 1)
Outputs:
select * from `users` where not `id` = 1
```

Grouped Chain:

```
knex('users').whereNot(function() {
  this.where('id', 1).orWhereNot('id', '>', 10)
}).orWhereNot({name: 'Tester'})
Outputs:
select * from `users` where not (`id` = 1 or not `id` > 10) or not `name` = 'Tester'
```

Operator:

```
knex('users').whereNot('votes', '>', 100)
Outputs:
select * from `users` where not `votes` > 100
```

CAVEAT: WhereNot is not suitable for "in" and "between" type subqueries. You should use "not in" and "not between" instead.

```
const subquery = knex('users')
  .whereNot('votes', '>', 100)
  .andWhere('status', 'active')
  .orWhere('name', 'John')
  .select('id');

knex('accounts').where('id', 'not in', subquery)
Outputs:
select * from `accounts` where `id` not in (select `id` from `users` where not `votes` > 100 and `status` = 'active' or `name` = 'John')
```

whereIn — .whereIn(column|columns, array|callback|builder) / .orWhereIn

Shorthand for .where('id', 'in', obj), the .whereIn and .orWhereIn methods add a "where in" clause to the query. Note that passing empty array as the value results in a query that never returns any rows (`WHERE 1 = 0`)

```
knex.select('name').from('users')
  .whereIn('id', [1, 2, 3])
  .orWhereIn('id', [4, 5, 6])
Outputs:
select `name` from `users` where `id` in (1, 2, 3) or `id` in (4, 5, 6)

knex.select('name').from('users')
  .whereIn('account_id', function() {
    this.select('id').from('accounts');
  })
Outputs:
select `name` from `users` where `account_id` in (select `id` from `accounts`)

const subquery = knex.select('id').from('accounts');

knex.select('name').from('users')
  .whereIn('account_id', subquery)
Outputs:
select `name` from `users` where `account_id` in (select `id` from `accounts`)

knex.select('name').from('users')
  .whereIn(['account_id', 'email'], [[3, 'test3@example.com'], [4, 'test4@example.com']])
Outputs:
select `name` from `users` where (`account_id`, `email`) in ((3, 'test3@example.com'), (4, 'test4@example.com'))

knex.select('name').from('users')
  .whereIn(['account_id', 'email'], knex.select('id', 'email').from('accounts'))
Outputs:
select `name` from `users` where (`account_id`, `email`) in (select `id`, `email` from `accounts`)
```

whereNotIn — .whereNotIn(column, array|callback|builder) / .orWhereNotIn

```
knex('users').whereNotIn('id', [1, 2, 3])
Outputs:
select * from `users` where `id` not in (1, 2, 3)

knex('users').where('name', 'like', '%Test%').orWhereNotIn('id', [1, 2, 3])
Outputs:
select * from `users` where `name` like '%Test%' or `id` not in (1, 2, 3)
```

whereNull — .whereNull(column) / .orWhereNull

```
knex('users').whereNull('updated_at')
Outputs:
select * from `users` where `updated_at` is null
```

whereNotNull — .whereNotNull(column) / .orWhereNotNull

```
knex('users').whereNotNull('created_at')
Outputs:
select * from `users` where `created_at` is not null
```

whereExists — .whereExists(builder | callback) / .orWhereExists

```
knex('users').whereExists(function() {
  this.select('*').from('accounts').whereRaw('users.account_id = accounts.id');
})
Outputs:
select * from `users` where exists (select * from `accounts` where users.account_id = accounts.id)

knex('users').whereExists(knex.select('*').from('accounts').whereRaw('users.account_id = accounts.id'))
Outputs:
select * from `users` where exists (select * from `accounts` where users.account_id = accounts.id)
```

whereNotExists — `.whereNotExists(builder | callback) / .orWhereNotExists`

```
knex('users').whereNotExists(function() {
  this.select('*').from('accounts').whereRaw('users.account_id = accounts.id');
})
Outputs:
select * from `users` where not exists (select * from `accounts` where users.account_id = accounts.id)

knex('users').whereNotExists(knex.select('*').from('accounts').whereRaw('users.account_id = accounts.id'))
Outputs:
select * from `users` where not exists (select * from `accounts` where users.account_id = accounts.id)
```

whereBetween — `.whereBetween(column, range) / .orWhereBetween`

```
knex('users').whereBetween('votes', [1, 100])
Outputs:
select * from `users` where `votes` between 1 and 100
```

whereNotBetween — `.whereNotBetween(column, range) / .orWhereNotBetween`

```
knex('users').whereNotBetween('votes', [1, 100])
Outputs:
select * from `users` where `votes` not between 1 and 100
```

whereRaw — `.whereRaw(query, [bindings])`

Convenience helper for .where(knex.raw(query)).

```
knex('users').whereRaw('id = ?', [1])
Outputs:
select * from `users` where id = 1
```

whereLike — `.whereLike(column, string|builder|raw)`

Adds a where clause with case-sensitive substring comparison on a given column with a given value.

```
knex('users').whereLike('email', '%mail%')
Outputs:
select * from `users` where `email` like '%mail%' COLLATE utf8_bin
```

whereILike — `.whereILike(column, string|builder|raw)`

Adds a where clause with case-insensitive substring comparison on a given column with a given value.

```
knex('users').whereILike('email', '%mail%')
Outputs:
select * from `users` where `email` like '%mail%'
```

whereJsonObject — `.whereJsonObject(column, string|json|builder|raw)`

Adds a where clause with json object comparison on given json column.

```
knex('users').whereJsonObject('json_col', { "name" : "user_name"})
Outputs:
select * from `users` where json_contains(`json_col`, '{\"name\":\"user_name\"}')
```

whereJsonPath — `.whereJsonPath(column, jsonPath, operator, value)`

Adds a where clause with comparison of a value returned by a JsonPath given an operator and a value.

```
knex('users').whereJsonPath('json_col', '$.age', '>', 18)
Outputs:
select * from `users` where json_extract(`json_col`, '$.age') > 18

knex('users').whereJsonPath('json_col', '$.name', '=', 'username')
Outputs:
select * from `users` where json_extract(`json_col`, '$.name') = 'username'
```

whereJsonSupersetOf — `.whereJsonSupersetOf(column, string|json|builder|raw)`

Adds a where clause where the comparison is true if a json given by the column include a given value. Only on MySQL, PostgreSQL and CockroachDB.

```
knex('users').whereJsonSupersetOf('hobbies', { "sport" : "foot" })
Outputs:
select * from `users` where json_contains(`hobbies`,'{\"sport\":\"foot\"}')
```

whereJsonSubsetOf — `.whereJsonSubsetOf(column, string|json|builder|raw)`

Adds a where clause where the comparison is true if a json given by the column is included in a given value. Only on MySQL, PostgreSQL and CockroachDB.

```
// given a hobby column with { "sport" : "tennis" }, the where clause is true
knex('users').whereJsonSubsetOf('hobby', { "sport" : "tennis", "book" : "fantasy" })
Outputs:
select * from `users` where json_contains('{\"sport\":\"tennis\",\"book\":\"fantasy\"}',`hobby`)
```

Join Methods

Several methods are provided which assist in building joins.

join — `.join(table, first, [operator], second)`

The join builder can be used to specify joins between tables, with the first argument being the joining table, the next three arguments being the first join column, the join operator and the second join column, respectively.

```
knex('users')
  .join('contacts', 'users.id', '=', 'contacts.user_id')
  .select('users.id', 'contacts.phone')
Outputs:
select `users`.`id`, `contacts`.`phone` from `users` inner join `contacts` on `users`.`id` = `contacts`.`user_id`
```

```
knex('users')
  .join('contacts', 'users.id', 'contacts.user_id')
  .select('users.id', 'contacts.phone')
Outputs:
select `users`.`id`, `contacts`.`phone` from `users` inner join `contacts` on `users`.`id` = `contacts`.`user_id`
```

For grouped joins, specify a function as the second argument for the join query, and use `on` with `orOn` or `andOn` to create joins that are grouped with parentheses.

```
knex.select('*').from('users').join('accounts', function() {
  this.on('accounts.id', '=', 'users.account_id').orOn('accounts.owner_id', '=', 'users.id')
})
Outputs:
select * from `users` inner join `accounts` on `accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`
```

For nested join statements, specify a function as first argument of `on`, `orOn` or `andOn`

```
knex.select('*').from('users').join('accounts', function() {
  this.on(function() {
    this.on('accounts.id', '=', 'users.account_id')
    this.orOn('accounts.owner_id', '=', 'users.id')
  })
})
Outputs:
select * from `users` inner join `accounts` on (`accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`)
```

It is also possible to use an object to represent the join syntax.

```
knex.select('*').from('users').join('accounts', {'accounts.id': 'users.account_id'})
Outputs:
select * from `users` inner join `accounts` on `accounts`.`id` = `users`.`account_id`
```

If you need to use a literal value (string, number, or boolean) in a join instead of a column, use `knex.raw`.

```
knex.select('*').from('users').join('accounts', 'accounts.type', knex.raw('?', ['admin']))
Outputs:
select * from `users` inner join `accounts` on `accounts`.`type` = 'admin'
```

innerJoin — `.innerJoin(table, ~mixed~)`

```
knex.from('users').innerJoin('accounts', 'users.id', 'accounts.user_id')
Outputs:
select * from `users` inner join `accounts` on `users`.`id` = `accounts`.`user_id`
```

```
knex.table('users').innerJoin('accounts', 'users.id', '=', 'accounts.user_id')
Outputs:
select * from `users` inner join `accounts` on `users`.`id` = `accounts`.`user_id`
```

```
knex('users').innerJoin('accounts', function() {
  this.on('accounts.id', '=', 'users.account_id').orOn('accounts.owner_id', '=', 'users.id')
})
Outputs:
select * from `users` inner join `accounts` on `accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`
```

leftJoin — `.leftJoin(table, ~mixed~)`

```
knex.select('*').from('users').leftJoin('accounts', 'users.id', 'accounts.user_id')
Outputs:
select * from `users` left join `accounts` on `users`.`id` = `accounts`.`user_id`
```

```
knex.select('*').from('users').leftJoin('accounts', function() {
  this.on('accounts.id', '=', 'users.account_id').orOn('accounts.owner_id', '=', 'users.id')
})
Outputs:
select * from `users` left join `accounts` on `accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`
```

leftOuterJoin — `.leftOuterJoin(table, ~mixed~)`

```
knex.select('*').from('users').leftOuterJoin('accounts', 'users.id', 'accounts.user_id')
Outputs:
select * from `users` left outer join `accounts` on `users`.`id` = `accounts`.`user_id`
```

```
knex.select('*').from('users').leftOuterJoin('accounts', function() {
  this.on('accounts.id', '=', 'users.account_id').orOn('accounts.owner_id', '=', 'users.id')
})
Outputs:
select * from `users` left outer join `accounts` on `accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`
```

rightJoin — `.rightJoin(table, ~mixed~)`

```
knex.select('*').from('users').rightJoin('accounts', 'users.id', 'accounts.user_id')
Outputs:
select * from `users` right join `accounts` on `users`.`id` = `accounts`.`user_id`
```

```
knex.select('*').from('users').rightJoin('accounts', function() {
  this.on('accounts.id', '=', 'users.account_id').orOn('accounts.owner_id', '=', 'users.id')
})
Outputs:
select * from `users` right join `accounts` on `accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`
```

rightOuterJoin — `.rightOuterJoin(table, ~mixed~)`

```
knex.select('*').from('users').rightOuterJoin('accounts', 'users.id', 'accounts.user_id')
Outputs:
select * from `users` right outer join `accounts` on `users`.`id` = `accounts`.`user_id`
```

```
knex.select('*').from('users').rightOuterJoin('accounts', function() {
  this.on('accounts.id', '=', 'users.account_id').orOn('accounts.owner_id', '=', 'users.id')
})
Outputs:
select * from `users` right outer join `accounts` on `accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`
```

fullOuterJoin — `.fullOuterJoin(table, ~mixed~)`

```
knex.select('*').from('users').fullOuterJoin('accounts', 'users.id', 'accounts.user_id')
Outputs:
select * from `users` full outer join `accounts` on `users`.`id` = `accounts`.`user_id`

knex.select('*').from('users').fullOuterJoin('accounts', function() {
  this.on('accounts.id', '=', 'users.account_id').orOn('accounts.owner_id', '=', 'users.id')
})
Outputs:
select * from `users` full outer join `accounts` on `accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`
```

crossJoin — `.crossJoin(table, ~mixed~)`

Cross join conditions are only supported in MySQL and SQLite3. For join conditions rather use innerJoin.

```
knex.select('*').from('users').crossJoin('accounts')
Outputs:
select * from `users` cross join `accounts`

knex.select('*').from('users').crossJoin('accounts', 'users.id', 'accounts.user_id')
Outputs:
select * from `users` cross join `accounts` on `users`.`id` = `accounts`.`user_id`

knex.select('*').from('users').crossJoin('accounts', function() {
  this.on('accounts.id', '=', 'users.account_id').orOn('accounts.owner_id', '=', 'users.id')
})
Outputs:
select * from `users` cross join `accounts` on `accounts`.`id` = `users`.`account_id` or `accounts`.`owner_id` = `users`.`id`
```

joinRaw — `.joinRaw(sql, [bindings])`

```
knex.select('*').from('accounts').joinRaw('natural full join table1').where('id', 1)
Outputs:
select * from `accounts` natural full join table1 where `id` = 1

knex.select('*').from('accounts').join(knex.raw('natural full join table1')).where('id', 1)
Outputs:
select * from `accounts` inner join natural full join table1 where `id` = 1
```

OnClauses

onIn — `.onIn(column, values)`

Adds a onIn clause to the query.

```
knex.select('*').from('users').join('contacts', function() {
  this.on('users.id', '=', 'contacts.id').onIn('contacts.id', [7, 15, 23, 41])
})
Outputs:
select * from `users` inner join `contacts` on `users`.`id` = `contacts`.`id` and `contacts`.`id` in (7, 15, 23, 41)
```

onNotIn — `.onNotIn(column, values)`

Adds a onNotIn clause to the query.

```
knex.select('*').from('users').join('contacts', function() {
  this.on('users.id', '=', 'contacts.id').onNotIn('contacts.id', [7, 15, 23, 41])
})
Outputs:
select * from `users` inner join `contacts` on `users`.`id` = `contacts`.`id` and `contacts`.`id` not in (7, 15, 23, 41)
```

onNull — `.onNull(column)`

Adds a onNull clause to the query.

```
knex.select('*').from('users').join('contacts', function() {
  this.on('users.id', '=', 'contacts.id').onNull('contacts.email')
})
Outputs:
select * from `users` inner join `contacts` on `users`.`id` = `contacts`.`id` and `contacts`.`email` is null
```

onNotNull — `.onNotNull(column)`

Adds a onNotNull clause to the query.

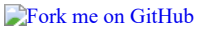
```
knex.select('*').from('users').join('contacts', function() {
  this.on('users.id', '=', 'contacts.id').onNotNull('contacts.email')
})
Outputs:
select * from `users` inner join `contacts` on `users`.`id` = `contacts`.`id` and `contacts`.`email` is not null
```

onExists — `.onExists(builder | callback)`

Adds a onExists clause to the query.

```
knex.select('*').from('users').join('contacts', function() {
  this.on('users.id', '=', 'contacts.id').onExists(function() {
    this.select('*').from('accounts').whereRaw('users.account_id = accounts.id');
  })
})
Outputs:
select * from `users` inner join `contacts` on `users`.`id` = `contacts`.`id` and exists (select * from `accounts` where users.account_id = accounts.
```

onNotExists — `.onNotExists(builder | callback)`



Adds a onNotExists clause to the query.

```
knex.select('*').from('users').join('contacts', function() {
  this.on('users.id', '=', 'contacts.id').onNotExists(function() {
    this.select('*').from('accounts').whereRaw('users.account_id = accounts.id');
  })
})
Outputs:
select * from `users` inner join `contacts` on `users`.`id` = `contacts`.`id` and not exists (select * from `accounts` where users.account_id = accour
```

onBetween — `.onBetween(column, range)`

Adds a onBetween clause to the query.

```
knex.select('*').from('users').join('contacts', function() {
  this.on('users.id', '=', 'contacts.id').onBetween('contacts.id', [5, 30])
})
Outputs:
select * from `users` inner join `contacts` on `users`.`id` = `contacts`.`id` and `contacts`.`id` between 5 and 30
```

onNotBetween — `.onNotBetween(column, range)`

Adds a onNotBetween clause to the query.

```
knex.select('*').from('users').join('contacts', function() {
  this.on('users.id', '=', 'contacts.id').onNotBetween('contacts.id', [5, 30])
})
Outputs:
select * from `users` inner join `contacts` on `users`.`id` = `contacts`.`id` and `contacts`.`id` not between 5 and 30
```

onJsonPathEquals — `.onJsonPathEquals(column, range)`

Adds a onJsonPathEquals clause to the query. The clause performs a join on value returned by two json paths on two json columns.

```
knex('cities')
  .select('cities.name as cityName', 'country.name as countryName')
  .join('country', function () {
    this.onJsonPathEquals(
      'country_name', // json column in cities
      '$.country.name', // json path to country name in 'country_name' column
      'description', // json column in country
      '$.name' // json field in 'description' column
    );
  })
Outputs:
select `cities`.`name` as `cityName`, `country`.`name` as `countryName` from `cities` inner join `country` on json_extract(`country_name`, '$.country
```

ClearClauses

clear — `.clear(statement)`

Clears the specified operator from the query. Available operators: 'select' alias 'columns', 'with', 'select', 'columns', 'where', 'union', 'join', 'group', 'order', 'having', 'limit', 'offset', 'counter', 'counters'. Counter(s) alias for method .clearCounter()

```
knex.select('email', 'name').from('users').where('id', '<', 10).clear('select').clear('where')
Outputs:
select * from `users`
```

clearSelect — `.clearSelect()`

Deprecated, use clear('select'). Clears all select clauses from the query, excluding subqueries.

```
knex.select('email', 'name').from('users').clearSelect()
Outputs:
select * from `users`
```

clearWhere — `.clearWhere()`

Deprecated, use clear('where'). Clears all where clauses from the query, excluding subqueries.

```
knex.select('email', 'name').from('users').where('id', 1).clearWhere()
Outputs:
select `email`, `name` from `users`
```

clearGroup — `.clearGroup()`

Deprecated, use clear('group'). Clears all group clauses from the query, excluding subqueries.

```
knex.select().from('users').groupBy('id').clearGroup()
Outputs:
select * from `users`
```

clearOrder — `.clearOrder()`

Deprecated, use clear('order'). Clears all order clauses from the query, excluding subqueries.

```
knex.select().from('users').orderBy('name', 'desc').clearOrder()
Outputs:
select * from `users`
```

clearHaving — `.clearHaving()`

Deprecated, use clear('having'). Clears all having clauses from the query, excluding subqueries.

```
knex.select().from('users').having('id', '>', 5).clearHaving()
Outputs:
select * from `users`
```

clearCounters — `.clearCounters()`

Clears all increments/decrements clauses from the query.

```
knex('accounts')
  .where('id', '=', 1)
  .update({ email: 'foo@bar.com' })
  .decrement({
    balance: 50,
  })
  .clearCounters()
Outputs:
update `accounts` set `email` = 'foo@bar.com' where `id` = 1
```

distinct — `.distinct([*columns])`

Sets a distinct clause on the query. If the parameter is falsy or empty array, method falls back to '*'.

```
// select distinct 'first_name' from customers
knex('customers')
  .distinct('first_name', 'last_name')
Outputs:
select distinct `first_name`, `last_name` from `customers`

// select which eliminates duplicate rows
knex('customers')
 .distinct()
Outputs:
select distinct * from `customers`
```

distinctOn — `.distinctOn([*columns])`

PostgreSQL only. Adds a distinctOn clause to the query.

```
knex('users').distinctOn('age')
Error:
.distinctOn() is currently only supported on PostgreSQL
```

groupBy — `.groupBy(*names)`

Adds a group by clause to the query.

```
knex('users').groupBy('count')
Outputs:
select * from `users` group by `count`
```

groupByRaw — `.groupByRaw(sql)`

Adds a raw group by clause to the query.

```
knex.select('year', knex.raw('SUM(profit)')).from('sales').groupByRaw('year WITH ROLLUP')
Outputs:
select `year`, SUM(profit) from `sales` group by year WITH ROLLUP
```

orderBy — `.orderBy(column|columns, [direction], [nulls])`

Adds an order by clause to the query. column can be string, or list mixed with string and object. nulls specify where the nulls values are put (can be 'first' or 'last').

Single Column:

```
knex('users').orderBy('email')
Outputs:
select * from `users` order by `email` asc

knex('users').orderBy('name', 'desc')
Outputs:
select * from `users` order by `name` desc

knex('users').orderBy('name', 'desc', 'first')
Outputs:
select * from `users` order by (`name` is not null) desc
```

Multiple Columns:

```
knex('users').orderBy(['email', { column: 'age', order: 'desc' }])
Outputs:
select * from `users` order by `email` asc, `age` desc

knex('users').orderBy([{ column: 'email' }, { column: 'age', order: 'desc' }])
Outputs:
select * from `users` order by `email` asc, `age` desc

knex('users').orderBy([{ column: 'email' }, { column: 'age', order: 'desc', nulls: 'last' }])
Outputs:
select * from `users` order by `email` asc, (`age` is null) desc
```

orderByRaw — `.orderByRaw(sql)`

Adds an order by raw clause to the query.

```
knex.select('*').from('table').orderByRaw('col DESC NULLS LAST')
Outputs:
select * from `table` order by col DESC NULLS LAST
```

Having Clauses

having — `.having(column, operator, value)`

Adds a having clause to the query.

```
knex('users')
  .groupBy('count')
```

```
  .orderBy('name', 'desc')
  .having('count', '>', 100)
Outputs:
select * from `users` group by `count` having `count` > 100 order by `name` desc
```

havingIn — `.havingIn(column, values)`

Adds a havingIn clause to the query.

```
knex.select('*').from('users').havingIn('id', [5, 3, 10, 17])
Outputs:
select * from `users` having `id` in (5, 3, 10, 17)
```

havingNotIn — `.havingNotIn(column, values)`

Adds a havingNotIn clause to the query.

```
knex.select('*').from('users').havingNotIn('id', [5, 3, 10, 17])
Outputs:
select * from `users` having `id` not in (5, 3, 10, 17)
```

havingNull — `.havingNull(column)`

Adds a havingNull clause to the query.

```
knex.select('*').from('users').havingNull('email')
Outputs:
select * from `users` having `email` is null
```

havingNotNull — `.havingNotNull(column)`

Adds a havingNotNull clause to the query.

```
knex.select('*').from('users').havingNotNull('email')
Outputs:
select * from `users` having `email` is not null
```

havingExists — `.havingExists(builder | callback)`

Adds a havingExists clause to the query.

```
knex.select('*').from('users').havingExists(function() {
  this.select('*').from('accounts').whereRaw('users.account_id = accounts.id');
})
Outputs:
select * from `users` having exists (select * from `accounts` where users.account_id = accounts.id)
```

havingNotExists — `.havingNotExists(builder | callback)`

Adds a havingNotExists clause to the query.

```
knex.select('*').from('users').havingNotExists(function() {
  this.select('*').from('accounts').whereRaw('users.account_id = accounts.id');
})
Outputs:
select * from `users` having not exists (select * from `accounts` where users.account_id = accounts.id)
```

havingBetween — `.havingBetween(column, range)`

Adds a havingBetween clause to the query.

```
knex.select('*').from('users').havingBetween('id', [5, 10])
Outputs:
select * from `users` having `id` between 5 and 10
```

havingNotBetween — `.havingNotBetween(column, range)`

Adds a havingNotBetween clause to the query.

```
knex.select('*').from('users').havingNotBetween('id', [5, 10])
Outputs:
select * from `users` having `id` not between 5 and 10
```

havingRaw — `.havingRaw(sql, [bindings])`

Adds a havingRaw clause to the query.

```
knex('users')
  .groupBy('count')
  .orderBy('name', 'desc')
  .havingRaw('count > ?', [100])
Outputs:
select * from `users` group by `count` having count > 100 order by `name` desc
```

offset — `.offset(value, options={skipBinding: boolean})`

Adds an offset clause to the query. An optional skipBinding parameter may be specified which would avoid setting offset as a prepared value (some databases don't allow prepared values for offset).

```
knex.select('*').from('users').offset(10)
Outputs:
select * from `users` limit 18446744073709551615 offset 10

knex.select('*').from('users').offset(10).toSQL().sql
Outputs:
select * from `users` limit 18446744073709551615 offset ?

// Offset value isn't a prepared value.
knex.select('*').from('users').offset(10, {skipBinding: true}).toSQL().sql
```

```
Outputs:
select * from `users` limit 18446744073709551615 offset 10
```

limit — `.limit(value, options={skipBinding: boolean})`

Adds a limit clause to the query. An optional skipBinding parameter may be specified to avoid adding limit as a prepared value (some databases don't allow prepared values for limit).

```
knex.select('*').from('users').limit(10).offset(30)
Outputs:
select * from `users` limit 10 offset 30
```

```
knex.select('*').from('users').limit(10).offset(30).toSQL().sql
Outputs:
select * from `users` limit ? offset ?
```

```
// Limit value isn't a prepared value.
knex.select('*').from('users').limit(10, {skipBinding: true}).offset(30).toSQL().sql
Outputs:
select * from `users` limit 10 offset ?
```

union — `.union([*queries], [wrap])`

Creates a union query, taking an array or a list of callbacks, builders, or raw statements to build the union statement, with optional boolean wrap. If the `wrap` parameter is `true`, the queries will be individually wrapped in parentheses.

```
knex.select('*').from('users').whereNull('last_name').union(function() {
  this.select('*').from('users').whereNull('first_name')
})
Outputs:
select * from `users` where `last_name` is null union select * from `users` where `first_name` is null
```

```
knex.select('*').from('users').whereNull('last_name').union([
  knex.select('*').from('users').whereNull('first_name')
])
Outputs:
select * from `users` where `last_name` is null union select * from `users` where `first_name` is null
```

```
knex.select('*').from('users').whereNull('last_name').union(
  knex.raw('select * from users where first_name is null'),
  knex.raw('select * from users where email is null')
)
Outputs:
select * from `users` where `last_name` is null union select * from users where first_name is null union select * from users where email is null
```

unionAll — `.unionAll([*queries], [wrap])`

Creates a union all query, with the same method signature as the union method. If the `wrap` parameter is `true`, the queries will be individually wrapped in parentheses.

```
knex.select('*').from('users').whereNull('last_name').unionAll(function() {
  this.select('*').from('users').whereNull('first_name');
})
Outputs:
select * from `users` where `last_name` is null union all select * from `users` where `first_name` is null
```

```
knex.select('*').from('users').whereNull('last_name').unionAll([
  knex.select('*').from('users').whereNull('first_name')
])
Outputs:
select * from `users` where `last_name` is null union all select * from `users` where `first_name` is null
```

```
knex.select('*').from('users').whereNull('last_name').unionAll(
  knex.raw('select * from users where first_name is null'),
  knex.raw('select * from users where email is null')
)
Outputs:
select * from `users` where `last_name` is null union all select * from users where first_name is null union all select * from users where email is nu
```

intersect — `.intersect([*queries], [wrap])`

Creates an intersect query, taking an array or a list of callbacks, builders, or raw statements to build the intersect statement, with optional boolean wrap. If the `wrap` parameter is `true`, the queries will be individually wrapped in parentheses. The intersect method is unsupported on MySQL.

```
knex.select('*').from('users').whereNull('last_name').intersect(function() {
  this.select('*').from('users').whereNull('first_name')
})
Outputs:
select * from `users` where `last_name` is null intersect select * from `users` where `first_name` is null
```

```
knex.select('*').from('users').whereNull('last_name').intersect([
  knex.select('*').from('users').whereNull('first_name')
])
Outputs:
select * from `users` where `last_name` is null intersect select * from `users` where `first_name` is null
```

```
knex.select('*').from('users').whereNull('last_name').intersect(
  knex.raw('select * from users where first_name is null'),
  knex.raw('select * from users where email is null')
)
Outputs:
select * from `users` where `last_name` is null intersect select * from users where first_name is null intersect select * from users where email is nu
```

insert — `.insert(data, [returning], [options])`

Creates an insert query, taking either a hash of properties to be inserted into the row, or an array of inserts, to be executed as a single insert command. If returning array is passed e.g. ['id', 'title'], it resolves the promise / fulfills the callback with an array of all the added rows with specified columns. It's a shortcut for returning method

```
// Returns [1] in "mysql", "sqlite", "oracle"; [] in "postgresql" unless the 'returning' parameter is set.
knex('books').insert({title: 'Slaughterhouse Five'})
Outputs:
insert into `books` (`title`) values ('Slaughterhouse Five')
```

```
// Normalizes for empty keys on multi-row insert:
knex('coords').insert([{x: 20}, {y: 30},  {x: 10, y: 20}])
Outputs:
insert into `coords` (`x`, `y`) values (20, DEFAULT), (DEFAULT, 30), (10, 20)

// Returns [2] in "mysql", "sqlite"; [2, 3] in "postgresql"
knex.insert([{title: 'Great Gatsby'}, {title: 'Fahrenheit 451'}], ['id']).into('books')
Outputs:
insert into `books` (`title`) values ('Great Gatsby'), ('Fahrenheit 451')
```

For MSSQL, triggers on tables can interrupt returning a valid value from the standard insert statements. You can add the `includeTriggerModifications` option to get around this issue. This modifies the SQL so the proper values can be returned. This only modifies the statement if you are using MSSQL, a returning value is specified, and the `includeTriggerModifications` option is set.

```
// Adding the option includeTriggerModifications allows you to
// run statements on tables that contain triggers. Only affects MSSQL.
knex('books')
  .insert({title: 'Alice in Wonderland'}, ['id'], { includeTriggerModifications: true })
```

If one prefers that undefined keys are replaced with `NULL` instead of `DEFAULT` one may give `useNullAsDefault` configuration parameter in knex config.

```
const knex = require('knex')({
  client: 'mysql',
  connection: {
    host : '127.0.0.1',
    port : 3306,
    user : 'your_database_user',
    password : 'your_database_password',
    database : 'myapp_test'
  },
  useNullAsDefault: true
});

knex('coords').insert([{x: 20}, {y: 30}, {x: 10, y: 20}])
// insert into `coords` (`x`, `y`) values (20, NULL), (NULL, 30), (10, 20)"
```

onConflict — `insert(..).onConflict(column) / insert(..).onConflict([column1, column2, ...]) / insert(..).onConflict(knex.raw(...))`

Implemented for the PostgreSQL, MySQL, and SQLite databases. A modifier for insert queries that specifies alternative behaviour in the case of a conflict. A conflict occurs when a table has a PRIMARY KEY or a UNIQUE index on a column (or a composite index on a set of columns) and a row being inserted has the same value as a row which already exists in the table in those column(s). The default behaviour in case of conflict is to raise an error and abort the query. Using this method you can change this behaviour to either silently ignore the error by using .onConflict().ignore() or to update the existing row with new data (perform an "UPSERT") by using .onConflict().merge().

Note: For PostgreSQL and SQLite, the column(s) specified by this method must either be the table's PRIMARY KEY or have a UNIQUE index on them, or the query will fail to execute. When specifying multiple columns, they must be a composite PRIMARY KEY or have composite UNIQUE index. MySQL will ignore the specified columns and always use the table's PRIMARY KEY. For cross-platform support across PostgreSQL, MySQL, and SQLite you must both explicitly specify the columns in .onConflict() and those column(s) must be the table's PRIMARY KEY.

For PostgreSQL and SQLite, you can use knex.raw(...) function in onConflict. It can be useful to specify condition when you have partial index :

```
knex('tableName')
  .insert({
    email: "ignore@example.com",
    name: "John Doe",
    active: true
  })
   // ignore only on email conflict and active is true.
  .onConflict(knex.raw('(email) where active'))
  .ignore()
```

See documentation on .ignore() and .merge() methods for more details.

ignore — `insert(..).onConflict(..).ignore()`

Implemented for the PostgreSQL, MySQL, and SQLite databases. Modifies an insert query, and causes it to be silently dropped without an error if a conflict occurs. Uses INSERT IGNORE in MySQL, and adds an ON CONFLICT (columns) DO NOTHING clause to the insert statement in PostgreSQL and SQLite.

```
knex('tableName')
  .insert({
    email: "ignore@example.com",
    name: "John Doe"
  })
  .onConflict('email')
  .ignore()
Outputs:
insert ignore into `tableName` (`email`, `name`) values ('ignore@example.com', 'John Doe')
```

merge — `insert(..).onConflict(..).merge() / insert(..).onConflict(..).merge(updates)`

Implemented for the PostgreSQL, MySQL, and SQLite databases. Modifies an insert query, to turn it into an 'upsert' operation. Uses ON DUPLICATE KEY UPDATE in MySQL, and adds an ON CONFLICT (columns) DO UPDATE clause to the insert statement in PostgreSQL and SQLite. By default, it merges all columns.

```
knex('tableName')
  .insert({
    email: "ignore@example.com",
    name: "John Doe"
  })
  .onConflict('email')
  .merge()
Outputs:
insert into `tableName` (`email`, `name`) values ('ignore@example.com', 'John Doe') on duplicate key update `email` = values(`email`), `name` = value
```

This also works with batch inserts:

```
knex('tableName')
  .insert([
    { email: "john@example.com", name: "John Doe" },
```

```
    { email: "jane@example.com", name: "Jane Doe" },
    { email: "alex@example.com", name: "Alex Doe" },
  ])
  .onConflict('email')
  .merge()
Outputs:
insert into `tableName` (`email`, `name`) values ('john@example.com', 'John Doe'), ('jane@example.com', 'Jane Doe'), ('alex@example.com', 'Alex Doe')
```

It is also possible to specify a subset of the columns to merge when a conflict occurs. For example, you may want to set a 'created_at' column when inserting but would prefer not to update it if the row already exists:

```
const timestamp = Date.now();
knex('tableName')
  .insert({
    email: "ignore@example.com",
    name: "John Doe",
    created_at: timestamp,
    updated_at: timestamp,
  })
  .onConflict('email')
  .merge(['email', 'name', 'updated_at'])
```

It is also possible to specify data to update separately from the data to insert. This is useful if you want to update with different data to the insert. For example, you may want to change a value if the row already exists:

```
const timestamp = Date.now();
knex('tableName')
  .insert({
    email: "ignore@example.com",
    name: "John Doe",
    created_at: timestamp,
    updated_at: timestamp,
  })
  .onConflict('email')
  .merge({
    name: "John Doe The Second",
  })
```

For PostgreSQL/SQLite databases only, it is also possible to add a WHERE clause to conditionally update only the matching rows:

```
const timestamp = Date.now();
knex('tableName')
  .insert({
    email: "ignore@example.com",
    name: "John Doe",
    created_at: timestamp,
    updated_at: timestamp,
  })
  .onConflict('email')
  .merge({
    name: "John Doe",
    updated_at: timestamp,
  })
  .where('updated_at', '<', timestamp)
```

upsert — `.upsert(data, [returning], [options])`

Implemented for the CockroachDB. Creates an upsert query, taking either a hash of properties to be inserted into the row, or an array of upserts, to be executed as a single upsert command. If returning array is passed e.g. ['id', 'title'], it resolves the promise / fulfills the callback with an array of all the added rows with specified columns. It's a shortcut for returning method

```
// insert new row with unique index on title column
knex('books').upsert({title: 'Great Gatsby'})
Error:
Upsert is not yet supported for dialect mysql

// update row by unique title 'Great Gatsby' and insert row with title 'Fahrenheit 451'
knex('books').upsert([{title: 'Great Gatsby'}, {title: 'Fahrenheit 451'}], ['id'])
Error:
Upsert is not yet supported for dialect mysql

// Normalizes for empty keys on multi-row upsert, result sql: ("x", "y") values (20, default), (default, 30), (10, 20):
knex('coords').upsert([{x: 20}, {y: 30}, {x: 10, y: 20}])
Error:
Upsert is not yet supported for dialect mysql
```

update — `.update(data, [returning], [options]) / .update(key, value, [returning], [options])`

Creates an update query, taking a hash of properties or a key/value pair to be updated based on the other query constraints. If returning array is passed e.g. ['id', 'title'], it resolves the promise / fulfills the callback with an array of all the updated rows with specified columns. It's a shortcut for returning method

```
knex('books')
  .where('published_date', '<', 2000)
  .update({
    status: 'archived',
    thisKeyIsSkipped: undefined
  })
Outputs:
update `books` set `status` = 'archived' where `published_date` < 2000

// Returns [1] in "mysql", "sqlite", "oracle"; [] in "postgresql" unless the 'returning' parameter is set.
knex('books').update('title', 'Slaughterhouse Five')
Outputs:
update `books` set `title` = 'Slaughterhouse Five'

// Returns [ { id: 42, title: "The Hitchhiker's Guide to the Galaxy" } ]
knex('books')
  .where({ id: 42 })
  .update({ title: "The Hitchhiker's Guide to the Galaxy" }, ['id', 'title'])
Outputs:
update `books` set `title` = 'The Hitchhiker\'s Guide to the Galaxy' where `id` = 42
```

For MSSQL, triggers on tables can interrupt returning a valid value from the standard update statements. You can add the `includeTriggerModifications` option to get around this issue. This modifies the SQL so the proper values can be returned. This only modifies the statement if you are using MSSQL, a returning value is specified, and the `includeTriggerModifications` option is set.

```
// Adding the option includeTriggerModifications allows you
// to run statements on tables that contain triggers. Only affects MSSQL.
knex('books')
  .update({title: 'Alice in Wonderland'}, ['id', 'title'], { includeTriggerModifications: true })
```

del / delete — `.del([returning], [options])`

Aliased to del as delete is a reserved word in JavaScript, this method deletes one or more rows, based on other conditions specified in the query. Resolves the promise / fulfills the callback with the number of affected rows for the query.

```
knex('accounts')
  .where('activated', false)
  .del()
Outputs:
delete from `accounts` where `activated` = false
```

For MSSQL, triggers on tables can interrupt returning a valid value from the standard delete statements. You can add the `includeTriggerModifications` option to get around this issue. This modifies the SQL so the proper values can be returned. This only modifies the statement if you are using MSSQL, a returning value is specified, and the `includeTriggerModifications` option is set.

```
// Adding the option includeTriggerModifications allows you
// to run statements on tables that contain triggers. Only affects MSSQL.
knex('books')
  .where('title', 'Alice in Wonderland')
  .del(['id', 'title'], { includeTriggerModifications: true })
```

For PostgreSQL, Delete statement with joins is both supported with classic 'join' syntax and 'using' syntax.

```
knex('accounts')
 .where('activated', false)
 .join('accounts', 'accounts.id', 'users.account_id')
 .del()
Outputs:
delete `accounts` from `accounts` inner join `accounts` on `accounts`.`id` = `users`.`account_id` where `activated` = false
```

using — `.using(tableName|tableNames)`

Can be used to define in PostgreSQL a delete statement with joins with explicit 'using' syntax. Classic join syntax can be used too.

```
knex('accounts')
 .where('activated', false)
 .using('accounts')
 .whereRaw('accounts.id = users.account_id')
 .del()
Error:
'using' function is only available in PostgreSQL dialect with Delete statements.
```

returning — `.returning(column, [options]) / .returning([column1, column2, ...], [options])`

Utilized by PostgreSQL, MSSQL, and Oracle databases, the returning method specifies which column should be returned by the insert, update and delete methods. Passed column parameter may be a string or an array of strings. The SQL result be reported as an array of objects, each containing a single property for each of the specified columns. The returning method is not supported on Amazon Redshift.

```
// Returns [ { id: 1 } ]
knex('books')
  .returning('id')
  .insert({title: 'Slaughterhouse Five'})
Outputs:
insert into `books` (`title`) values ('Slaughterhouse Five')

// Returns [ { id: 2 } ] in "mysql", "sqlite"; [ { id: 2 }, { id: 3 } ] in "postgresql"
knex('books')
  .returning('id')
  .insert([{title: 'Great Gatsby'}, {title: 'Fahrenheit 451'}])
Outputs:
insert into `books` (`title`) values ('Great Gatsby'), ('Fahrenheit 451')

// Returns [ { id: 1, title: 'Slaughterhouse Five' } ]
knex('books')
  .returning(['id','title'])
  .insert({title: 'Slaughterhouse Five'})
Outputs:
insert into `books` (`title`) values ('Slaughterhouse Five')
```

For MSSQL, triggers on tables can interrupt returning a valid value from the standard DML statements. You can add the `includeTriggerModifications` option to get around this issue. This modifies the SQL so the proper values can be returned. This only modifies the statement if you are using MSSQL, a returning value is specified, and the `includeTriggerModifications` option is set.

```
// Adding the option includeTriggerModifications allows you
// to run statements on tables that contain triggers. Only affects MSSQL.
knex('books')
  .returning(['id','title'], { includeTriggerModifications: true })
  .insert({title: 'Slaughterhouse Five'})
```

transacting — `.transacting(transactionObj)`

Used by knex.transaction, the transacting method may be chained to any query and passed the object you wish to join the query as part of the transaction for.

```
const Promise = require('bluebird');
knex.transaction(function(trx) {
  knex('books').transacting(trx).insert({name: 'Old Books'})
    .then(function(resp) {
      const id = resp[0];
      return someExternalMethod(id, trx);
    })
    .then(trx.commit)
```

```
    .catch(trx.rollback);
})
.then(function(resp) {
  console.log('Transaction complete.');
})
.catch(function(err) {
  console.error(err);
});
```

forUpdate — `.transacting(t).forUpdate()`

Dynamically added after a transaction is specified, the forUpdate adds a FOR UPDATE in PostgreSQL and MySQL during a select statement. Not supported on Amazon Redshift due to lack of table locks.

```
knex('tableName')
  .transacting(trx)
  .forUpdate()
  .select('*')
Outputs:
select * from `tableName` for update
```

forShare — `.transacting(t).forShare()`

Dynamically added after a transaction is specified, the forShare adds a FOR SHARE in PostgreSQL and a LOCK IN SHARE MODE for MySQL during a select statement. Not supported on Amazon Redshift due to lack of table locks.

```
knex('tableName')
  .transacting(trx)
  .forShare()
  .select('*')
Outputs:
select * from `tableName` lock in share mode
```

forNoKeyUpdate — `.transacting(t).forNoKeyUpdate()`

Dynamically added after a transaction is specified, the forNoKeyUpdate adds a FOR NO KEY UPDATE in PostgreSQL.

```
knex('tableName')
  .transacting(trx)
  .forNoKeyUpdate()
  .select('*')
Error:
this[this.single.lock] is not a function
```

forKeyShare — `.transacting(t).forKeyShare()`

Dynamically added after a transaction is specified, the forKeyShare adds a FOR KEY SHARE in PostgreSQL.

```
knex('tableName')
  .transacting(trx)
  .forKeyShare()
  .select('*')
Error:
this[this.single.lock] is not a function
```

skipLocked — `.skipLocked()`

MySQL 8.0+, MariaDB-10.6+ and PostgreSQL 9.5+ only. This method can be used after a lock mode has been specified with either forUpdate or forShare, and will cause the query to skip any locked rows, returning an empty set if none are available.

```
knex('tableName')
  .select('*')
  .forUpdate()
  .skipLocked()
Outputs:
select * from `tableName` for update skip locked
```

noWait — `.noWait()`

MySQL 8.0+, MariaDB-10.3+ and PostgreSQL 9.5+ only. This method can be used after a lock mode has been specified with either forUpdate or forShare, and will cause the query to fail immediately if any selected rows are currently locked.

```
knex('tableName')
  .select('*')
  .forUpdate()
  .noWait()
Outputs:
select * from `tableName` for update nowait
```

count — `.count(column|columns|raw, [options])`

Performs a count on the specified column or array of columns (note that some drivers do not support multiple columns). Also accepts raw expressions. The value returned from count (and other aggregation queries) is an array of objects like: `[{'COUNT(*)': 1}]`. The actual keys are dialect specific, so usually we would want to specify an alias (Refer examples below). Note that in Postgres, count returns a bigint type which will be a String and not a Number (more info).

```
knex('users').count('active')
Outputs:
select count(`active`) from `users`

knex('users').count('active', {as: 'a'})
Outputs:
select count(`active`) as `a` from `users`

knex('users').count('active as a')
Outputs:
select count(`active`) as `a` from `users`

knex('users').count({ a: 'active' })
Outputs:
select count(`active`) as `a` from `users`
```

```
knex('users').count({ a: 'active', v: 'valid' })
Outputs:
select count(`active`) as `a`, count(`valid`) as `v` from `users`

knex('users').count('id', 'active')
Outputs:
select count(`id`) from `users`

knex('users').count({ count: ['id', 'active'] })
Outputs:
select count(`id`, `active`) as `count` from `users`

knex('users').count(knex.raw('??', ['active']))
Outputs:
select count(`active`) from `users`
```

Usage with TypeScript

The value of count will, by default, have type of `string | number`. This may be counter-intuitive but some connectors (eg. postgres) will automatically cast BigInt result to string when javascript's Number type is not large enough for the value.

```
knex('users').count('age') // Resolves to: Record<string, number | string>

knex('users').count({count: '*'}) // Resolves to { count?: string | number | undefined; }
```

Working with `string | number` can be inconvenient if you are not working with large tables. Two alternatives are available:

```
// Be explicit about what you want as a result:
knex('users').count<Record<string, number>>('age');

// Setup a one time declaration to make knex use number as result type for all
// count and countDistinct invocations (for any table)
declare module "knex/types/result" {
    interface Registry {
        Count: number;
    }
}
```

Use **countDistinct** to add a distinct expression inside the aggregate function.

```
knex('users').countDistinct('active')
Outputs:
select count(distinct `active`) from `users`
```

min — `.min(column|columns|raw, [options])`

Gets the minimum value for the specified column or array of columns (note that some drivers do not support multiple columns). Also accepts raw expressions.

```
knex('users').min('age')
Outputs:
select min(`age`) from `users`

knex('users').min('age', {as: 'a'})
Outputs:
select min(`age`) as `a` from `users`

knex('users').min('age as a')
Outputs:
select min(`age`) as `a` from `users`

knex('users').min({ a: 'age' })
Outputs:
select min(`age`) as `a` from `users`

knex('users').min({ a: 'age', b: 'experience' })
Outputs:
select min(`age`) as `a`, min(`experience`) as `b` from `users`

knex('users').min('age', 'logins')
Outputs:
select min(`age`) from `users`

knex('users').min({ min: ['age', 'logins'] })
Outputs:
select min(`age`, `logins`) as `min` from `users`

knex('users').min(knex.raw('??', ['age']))
Outputs:
select min(`age`) from `users`
```

max — `.max(column|columns|raw, [options])`

Gets the maximum value for the specified column or array of columns (note that some drivers do not support multiple columns). Also accepts raw expressions.

```
knex('users').max('age')
Outputs:
select max(`age`) from `users`

knex('users').max('age', {as: 'a'})
Outputs:
select max(`age`) as `a` from `users`

knex('users').max('age as a')
Outputs:
select max(`age`) as `a` from `users`

knex('users').max({ a: 'age' })
Outputs:
select max(`age`) as `a` from `users`

knex('users').max('age', 'logins')
Outputs:
select max(`age`) from `users`
```

```
knex('users').max({ max: ['age', 'logins'] })
Outputs:
select max(`age`, `logins`) as `max` from `users`

knex('users').max({ max: 'age', exp: 'experience' })
Outputs:
select max(`age`) as `max`, max(`experience`) as `exp` from `users`

knex('users').max(knex.raw('??', ['age']))
Outputs:
select max(`age`) from `users`
```

sum — `.sum(column|columns|raw)`

Retrieve the sum of the values of a given column or array of columns (note that some drivers do not support multiple columns). Also accepts raw expressions.

```
knex('users').sum('products')
Outputs:
select sum(`products`) from `users`

knex('users').sum('products as p')
Outputs:
select sum(`products`) as `p` from `users`

knex('users').sum({ p: 'products' })
Outputs:
select sum(`products`) as `p` from `users`

knex('users').sum('products', 'orders')
Outputs:
select sum(`products`) from `users`

knex('users').sum({ sum: ['products', 'orders'] })
Outputs:
select sum(`products`, `orders`) as `sum` from `users`

knex('users').sum(knex.raw('??', ['products']))
Outputs:
select sum(`products`) from `users`
```

Use **sumDistinct** to add a distinct expression inside the aggregate function.

```
knex('users').sumDistinct('products')
Outputs:
select sum(distinct `products`) from `users`
```

avg — `.avg(column|columns|raw)`

Retrieve the average of the values of a given column or array of columns (note that some drivers do not support multiple columns). Also accepts raw expressions.

```
knex('users').avg('age')
Outputs:
select avg(`age`) from `users`

knex('users').avg('age as a')
Outputs:
select avg(`age`) as `a` from `users`

knex('users').avg({ a: 'age' })
Outputs:
select avg(`age`) as `a` from `users`

knex('users').avg('age', 'logins')
Outputs:
select avg(`age`) from `users`

knex('users').avg({ avg: ['age', 'logins'] })
Outputs:
select avg(`age`, `logins`) as `avg` from `users`

knex('users').avg(knex.raw('??', ['age']))
Outputs:
select avg(`age`) from `users`
```

Use **avgDistinct** to add a distinct expression inside the aggregate function.

```
knex('users').avgDistinct('age')
Outputs:
select avg(distinct `age`) from `users`
```

increment — `.increment(column, amount)`

Increments a column value by the specified amount. Object syntax is supported for `column`.

```
knex('accounts')
  .where('userid', '=', 1)
  .increment('balance', 10)
Outputs:
update `accounts` set `balance` = `balance` + 10 where `userid` = 1

knex('accounts')
  .where('id', '=', 1)
  .increment({
    balance: 10,
    times: 1,
  })
Outputs:
update `accounts` set `balance` = `balance` + 10, `times` = `times` + 1 where `id` = 1
```

decrement — `.decrement(column, amount)`

Decrements a column value by the specified amount. Object syntax is supported for `column`.

```
knex('accounts').where('userid', '=', 1).decrement('balance', 5)
Outputs:
update `accounts` set `balance` = `balance` - 5 where `userid` = 1

knex('accounts')
  .where('id', '=', 1)
  .decrement({
    balance: 50,
  })
Outputs:
update `accounts` set `balance` = `balance` - 50 where `id` = 1
```

truncate — `.truncate()`

Truncates the current table.

```
knex('accounts').truncate()
Outputs:
truncate `accounts`
```

pluck — `.pluck(id)`

This will pluck the specified column from each row in your results, yielding a promise which resolves to the array of values selected.

```
knex.table('users').pluck('id').then(function(ids) { console.log(ids); });
```

first — `.first([columns])`

Similar to select, but only retrieves & resolves with the first record from the query.

```
knex.table('users').first('id', 'name').then(function(row) { console.log(row); });
```

hintComment — `.hintComment(hint|hints)`

Add hints to the query using comment-like syntax `/*+ ... */`. MySQL and Oracle use this syntax for optimizer hints. Also various DB proxies and routers use this syntax to pass hints to alter their behavior. In other dialects the hints are ignored as simple comments.

```
knex('accounts').where('userid', '=', 1).hintComment('NO_ICP(accounts)')
Outputs:
select /*+ NO_ICP(accounts) */ * from `accounts` where `userid` = 1
```

clone — `.clone()`

Clones the current query chain, useful for re-using partial query snippets in other queries without mutating the original.

denseRank — `.denseRank(alias, ~mixed~)`

Add a dense_rank() call to your query. For all the following queries, alias can be set to a falsy value if not needed.

String Syntax — .denseRank(alias, orderByClause, [partitionByClause]) :

```
knex('users').select('*').denseRank('alias_name', 'email', 'firstName')
Outputs:
select *, dense_rank() over (partition by `firstName` order by `email`) as alias_name from `users`
```

It also accepts arrays of strings as argument :

```
knex('users').select('*').denseRank('alias_name', ['email', 'address'], ['firstName', 'lastName'])
Outputs:
select *, dense_rank() over (partition by `firstName`, `lastName` order by `email`, `address`) as alias_name from `users`
```

Raw Syntax — .denseRank(alias, rawQuery) :

```
knex('users').select('*').denseRank('alias_name', knex.raw('order by ??', ['email']))
Outputs:
select *, dense_rank() over (order by `email`) as alias_name from `users`
```

Function Syntax — .denseRank(alias, function) :

Use orderBy() and partitionBy() (both chainable) to build your query :

```
knex('users').select('*').denseRank('alias_name', function() {
  this.orderBy('email').partitionBy('firstName')
})
Outputs:
select *, dense_rank() over (partition by `firstName` order by `email`) as alias_name from `users`
```

rank — `.rank(alias, ~mixed~)`

Add a rank() call to your query. For all the following queries, alias can be set to a falsy value if not needed.

String Syntax — .rank(alias, orderByClause, [partitionByClause]) :

```
knex('users').select('*').rank('alias_name', 'email', 'firstName')
Outputs:
select *, rank() over (partition by `firstName` order by `email`) as alias_name from `users`
```

It also accepts arrays of strings as argument :

```
knex('users').select('*').rank('alias_name', ['email', 'address'], ['firstName', 'lastName'])
Outputs:
select *, rank() over (partition by `firstName`, `lastName` order by `email`, `address`) as alias_name from `users`
```

Raw Syntax — .rank(alias, rawQuery) :

```
knex('users').select('*').rank('alias_name', knex.raw('order by ??', ['email']))
Outputs:
select *, rank() over (order by `email`) as alias_name from `users`
```

Function Syntax — .rank(alias, function) :

Use orderBy() and partitionBy() (both chainable) to build your query :

```
knex('users').select('*').rank('alias_name', function() {
  this.orderBy('email').partitionBy('firstName')
})
Outputs:
select *, rank() over (partition by `firstName` order by `email`) as alias_name from `users`
```

rowNumber — `.rowNumber(alias, ~mixed~)`

Add a row_number() call to your query. For all the following queries, alias can be set to a falsy value if not needed.

String Syntax — .rowNumber(alias, orderByClause, [partitionByClause]) :

```
knex('users').select('*').rowNumber('alias_name', 'email', 'firstName')
Outputs:
select *, row_number() over (partition by `firstName` order by `email`) as alias_name from `users`
```

It also accepts arrays of strings as argument :

```
knex('users').select('*').rowNumber('alias_name', ['email', 'address'], ['firstName', 'lastName'])
Outputs:
select *, row_number() over (partition by `firstName`, `lastName` order by `email`, `address`) as alias_name from `users`
```

Raw Syntax — .rowNumber(alias, rawQuery) :

```
knex('users').select('*').rowNumber('alias_name', knex.raw('order by ??', ['email']))
Outputs:
select *, row_number() over (order by `email`) as alias_name from `users`
```

Function Syntax — .rowNumber(alias, function) :

Use orderBy() and partitionBy() (both chainable) to build your query :

```
knex('users').select('*').rowNumber('alias_name', function() {
  this.orderBy('email').partitionBy('firstName')
})
Outputs:
select *, row_number() over (partition by `firstName` order by `email`) as alias_name from `users`
```

partitionBy — `.partitionBy(column, direction)`

Partitions rowNumber, denseRank, rank after a specific column or columns. If direction is not supplied it will default to ascending order.

No direction sort :

```
knex('users').select('*').rowNumber('alias_name', function() {
  this.partitionBy('firstName');
});
```

With direction sort :

```
knex('users').select('*').rowNumber('alias_name', function() {
  this.partitionBy('firstName', 'desc');
});
```

With multiobject :

```
knex('users').select('*').rowNumber('alias_name', function() {
  this.partitionBy([{ column: 'firstName', order: 'asc' }, { column: 'lastName', order: 'desc' }]);
});
```

modify — `.modify(fn, *arguments)`

Allows encapsulating and re-using query snippets and common behaviors as functions. The callback function should receive the query builder as its first argument, followed by the rest of the (optional) parameters passed to modify.

```
const withUserName = function(queryBuilder, foreignKey) {
  queryBuilder.leftJoin('users', foreignKey, 'users.id').select('users.user_name');
};
knex.table('articles').select('title', 'body').modify(withUserName, 'articles_user.id').then(function(article) {
  console.log(article.user_name);
});
```

columnInfo — `.columnInfo([columnName])`

Returns an object with the column info about the current table, or an individual column if one is passed, returning an object with the following keys:

- **defaultValue**: the default value for the column
- **type**: the column type
- **maxLength**: the max length set for the column
- **nullable**: whether the column may be null

```
knex('users').columnInfo().then(function(info) { // ... });
```

debug — `.debug([enabled])`

Overrides the global debug setting for the current query chain. If enabled is omitted, query debugging will be turned on.

connection — `.connection(dbConnection)`

The method sets the db connection to use for the query without using the connection pool. You should pass to it the same object that acquireConnection() for the corresponding driver returns

```
const Pool = require('pg-pool')
const pool = new Pool({ ... })
const connection = await pool.connect();
  try {
    return await knex.connection(connection); // knex here is a query builder with query already built
  } catch (error) {
    // Process error
  } finally {
    connection.release();
  }
```

options — `.options()`

Allows for mixing in additional options as defined by database client specific libraries:

```
knex('accounts as a1')
  .leftJoin('accounts as a2', function() {
    this.on('a1.email', '<>', 'a2.email');
  })
  .select(['a1.email', 'a2.email'])
  .where(knex.raw('a1.id = 1'))
  .options({ nestTables: true, rowMode: 'array' })
  .limit(2)
  .then(...
```

queryContext — `.queryContext(context)`

Allows for configuring a context to be passed to the wrapIdentifier and postProcessResponse hooks:

```
knex('accounts as a1')
  .queryContext({ foo: 'bar' })
  .select(['a1.email', 'a2.email'])
```

The context can be any kind of value and will be passed to the hooks without modification. However, note that **objects will be shallow-cloned** when a query builder instance is cloned, which means that they will contain all the properties of the original object but will not be the same object reference. This allows modifying the context for the cloned query builder instance.

Calling `queryContext` with no arguments will return any context configured for the query builder instance.

Extending Query Builder

Important: this feature is experimental and its API may change in the future.

It allows to add custom function to the Query Builder.

Example:

```
const Knex = require('knex');
Knex.QueryBuilder.extend('customSelect', function(value) {
  return this.select(this.client.raw(`${value} as value`));
});

const meaningOfLife = await knex('accounts')
  .customSelect(42);
```

If using TypeScript, you can extend the QueryBuilder interface with your custom method.

1. Create a `knex.d.ts` file inside a `@types` folder (or any other folder).

```
// knex.d.ts

import { Knex as KnexOriginal } from 'knex';

declare module 'knex' {
  namespace Knex {
    interface QueryBuilder {
      customSelect<TRecord, TResult>(value: number): KnexOriginal.QueryBuilder<TRecord, TResult>;
    }
  }
}
```

2. Add the new `@types` folder to `typeRoots` in your `tsconfig.json`.

```
// tsconfig.json

{
  "compilerOptions": {
    "typeRoots": [
      "node_modules/@types",
      "@types"
    ],
  }
}
```

Transactions

Transactions are an important feature of relational databases, as they allow correct recovery from failures and keep a database consistent even in cases of system failure. All queries within a transaction are executed on the same database connection, and run the entire set of queries as a single unit of work. Any failure will mean the database will rollback any queries executed on that connection to the pre-transaction state.

Transactions are handled by passing a handler function into `knex.transaction`. The handler function accepts a single argument, an object which may be used in two ways:

1. As the "promise aware" knex connection
2. As an object passed into a query with and eventually call commit or rollback.

Consider these two examples:

```
// Using trx as a query builder:
knex.transaction(function(trx) {

  const books = [
    {title: 'Canterbury Tales'},
    {title: 'Moby Dick'},
    {title: 'Hamlet'}
  ];

  return trx
    .insert({name: 'Old Books'}, 'id')
    .into('catalogues')
    .then(function(ids) {
      books.forEach((book) => book.catalogue_id = ids[0]);
      return trx('books').insert(books);
    });
})
.then(function(inserts) {
  console.log(inserts.length + ' new books saved.');
})
.catch(function(error) {
  // If we get here, that means that neither the 'Old Books' catalogues insert,
  // nor any of the books inserts will have taken place.
  console.error(error);
});
```

And then this example:

```
// Using trx as a transaction object:
knex.transaction(function(trx) {

  const books = [
    {title: 'Canterbury Tales'},
    {title: 'Moby Dick'},
    {title: 'Hamlet'}
  ];

  knex.insert({name: 'Old Books'}, 'id')
    .into('catalogues')
    .transacting(trx)
    .then(function(ids) {
      books.forEach((book) => book.catalogue_id = ids[0]);
      return knex('books').insert(books).transacting(trx);
    })
    .then(trx.commit)
    .catch(trx.rollback);
})
.then(function(inserts) {
  console.log(inserts.length + ' new books saved.');
})
.catch(function(error) {
  // If we get here, that means that neither the 'Old Books' catalogues insert,
  // nor any of the books inserts will have taken place.
  console.error(error);
});
```

Same example as above using await/async:

```
try {
  await knex.transaction(async trx => {

    const books = [
      {title: 'Canterbury Tales'},
      {title: 'Moby Dick'},
      {title: 'Hamlet'}
    ];

    const ids = await trx('catalogues')
      .insert({
        name: 'Old Books'
      }, 'id')

    books.forEach((book) => book.catalogue_id = ids[0])
    const inserts = await trx('books').insert(books)

    console.log(inserts.length + ' new books saved.')
  })
} catch (error) {
  // If we get here, that means that neither the 'Old Books' catalogues insert,
  // nor any of the books inserts will have taken place.
  console.error(error);
}
```

Same example as above using another await/async approach:

```
try {
  await knex.transaction(async trx => {

    const books = [
      {title: 'Canterbury Tales'},
      {title: 'Moby Dick'},
      {title: 'Hamlet'}
    ];

    const ids = await knex('catalogues')
      .insert({
        name: 'Old Books'
      }, 'id')
      .transacting(trx)

    books.forEach(book => book.catalogue_id = ids[0])
```

```
    await knex('books')
      .insert(books)
      .transacting(trx)

    console.log(inserts.length + ' new books saved.')
  })
} catch (error) {
  console.error(error);
}
```

Throwing an error directly from the transaction handler function automatically rolls back the transaction, same as returning a rejected promise.

Notice that if a promise is not returned within the handler, it is up to you to ensure `trx.commit`, or `trx.rollback` are called, otherwise the transaction connection will hang.

Calling `trx.rollback` will return a rejected Promise. If you don't pass any argument to `trx.rollback`, a generic `Error` object will be created and passed in to ensure the Promise always rejects with something.

Note that Amazon Redshift does not support savepoints in transactions.

In some cases you may prefer to create transaction but only execute statements in it later. In such case call method `transaction` without a handler function:

```
// Using trx as a transaction object:
const trx = await knex.transaction();

const books = [
  {title: 'Canterbury Tales'},
  {title: 'Moby Dick'},
  {title: 'Hamlet'}
];

trx('catalogues')
  .insert({name: 'Old Books'}, 'id')
  .then(function(ids) {
    books.forEach((book) => book.catalogue_id = ids[0]);
    return trx('books').insert(books);
  })
  .then(trx.commit)
  .catch(trx.rollback);
```

If you want to create a reusable transaction instance, but do not want to actually start it until it is used, you can create a transaction provider instance. It will start transaction after being called for the first time, and return same transaction on subsequent calls:

```
// Does not start a transaction yet
const trxProvider = knex.transactionProvider();

const books = [
  {title: 'Canterbury Tales'},
  {title: 'Moby Dick'},
  {title: 'Hamlet'}
];

// Starts a transaction
const trx = await trxProvider();
const ids = await trx('catalogues')
  .insert({name: 'Old Books'}, 'id')
books.forEach((book) => book.catalogue_id = ids[0]);
await  trx('books').insert(books);

// Reuses same transaction
const sameTrx = await trxProvider();
const ids2 = await sameTrx('catalogues')
  .insert({name: 'New Books'}, 'id')
books.forEach((book) => book.catalogue_id = ids2[0]);
await sameTrx('books').insert(books);
```

You can access the promise that gets resolved after transaction is rolled back explicitly by user or committed, or rejected if it gets rolled back by DB itself, when using either way of creating transaction, from field `executionPromise`:

```
const trxProvider = knex.transactionProvider();
const trx = await trxProvider();
const trxPromise = trx.executionPromise;

const trx2 = await knex.transaction();
const trx2Promise = trx2.executionPromise;

const trxInitPromise = new Promise(async (resolve, reject) => {
  knex.transaction((transaction) => {
    resolve(transaction);
  });
});
const trx3 = await trxInitPromise;
const trx3Promise = trx3.executionPromise;
```

You can check if a transaction has been committed or rolled back with the method `isCompleted`:

```
const trx = await knex.transaction();
trx.isCompleted(); // false
await trx.commit();
trx.isCompleted(); // true

const trx2 = knex.transactionProvider();
await trx2.rollback();
trx2.isCompleted(); // true
```

You can check the property `knex.isTransaction` to see if the current knex instance you are working with is a transaction.

In case you need to specify an isolation level for your transaction, you can use a config parameter `isolationLevel`. Not supported by oracle and sqlite, options are `read uncommitted`, `read committed`, `repeatable read`, `snapshot` (mssql only), `serializable`.

```
// Simple read skew example
const isolationLevel = 'read committed';
const trx = await knex.transaction({isolationLevel});
const result1 = await trx(tableName).select();
await knex(tableName).insert({ id: 1, value: 1 });
const result2 = await trx(tableName).select();
await trx.commit();
// result1 may or may not deep equal result2 depending on isolation level
```

Schema Builder

The `knex.schema` is a **getter function**, which returns a stateful object containing the query. Therefore be sure to obtain a new instance of the `knex.schema` for every query. These methods return promises.

withSchema — `knex.schema.withSchema([schemaName])`

Specifies the schema to be used when using the schema-building commands.

```
knex.schema.withSchema('public').createTable('users', function (table) {
  table.increments();
})
Outputs:
create table `public`.`users` (`id` int unsigned not null auto_increment primary key)
```

createTable — `knex.schema.createTable(tableName, callback)`

Creates a new table on the database, with a callback function to modify the table's structure, using the schema-building commands.

```
knex.schema.createTable('users', function (table) {
  table.increments();
  table.string('name');
  table.timestamps();
})
Outputs:
create table `users` (`id` int unsigned not null auto_increment primary key, `name` varchar(255), `created_at` datetime, `updated_at` datetime)
```

createTableLike — `knex.schema.createTableLike(tableName, tableNameToCopy, [callback])`

Creates a new table on the database based on another table. Copy only the structure : columns, keys and indexes (expected on SQL Server which only copy columns) and not the data. Callback function can be specified to add columns in the duplicated table.

```
knex.schema.createTableLike('new_users', 'users')
Outputs:
create table `new_users` like `users`
```

```
// "new_users" table contains columns of users and two new columns 'age' and 'last_name'.
knex.schema.createTableLike('new_users', 'users', (table) => {
  table.integer('age');
  table.string('last_name');
})
Outputs:
create table `new_users` like `users`;
alter table `new_users` add `age` int, add `last_name` varchar(255)
```

dropTable — `knex.schema.dropTable(tableName)`

Drops a table, specified by tableName.

```
knex.schema.dropTable('users')
Outputs:
drop table `users`
```

dropTableIfExists — `knex.schema.dropTableIfExists(tableName)`

Drops a table conditionally if the table exists, specified by tableName.

```
knex.schema.dropTableIfExists('users')
Outputs:
drop table if exists `users`
```

renameTable — `knex.schema.renameTable(from, to)`

Renames a table from a current tableName to another.

```
knex.schema.renameTable('users', 'old_users')
Outputs:
rename table `users` to `old_users`
```

hasTable — `knex.schema.hasTable(tableName)`

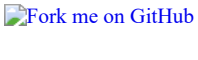
Checks for a table's existence by tableName, resolving with a boolean to signal if the table exists.

```
knex.schema.hasTable('users').then(function(exists) {
  if (!exists) {
    return knex.schema.createTable('users', function(t) {
      t.increments('id').primary();
      t.string('first_name', 100);
      t.string('last_name', 100);
      t.text('bio');
    });
  }
});
```

hasColumn — `knex.schema.hasColumn(tableName, columnName)`

Checks if a column exists in the current table, resolves the promise with a boolean, true if the column exists, false otherwise.

table — `knex.schema.table(tableName, callback)`

Chooses a database table, and then modifies the table, using the Schema Building functions inside of the callback. 

```
knex.schema.table('users', function (table) {
  table.dropColumn('name');
  table.string('first_name');
  table.string('last_name');
})
Outputs:
alter table `users` add `first_name` varchar(255), add `last_name` varchar(255);
alter table `users` drop `name`
```

alterTable — `knex.schema.alterTable(tableName, callback)`

Chooses a database table, and then modifies the table, using the Schema Building functions inside of the callback.

```
knex.schema.alterTable('users', function (table) {
  table.dropColumn('name');
  table.string('first_name');
  table.string('last_name');
})
Outputs:
alter table `users` add `first_name` varchar(255), add `last_name` varchar(255);
alter table `users` drop `name`
```

createView — `knex.schema.createView(tableName, callback)`

Creates a new view on the database, with a callback function to modify the view's structure, using the schema-building commands.

```
knex.schema.createView('users_view', function (view) {
  view.columns(['first_name']);
  view.as(knex('users').select('first_name').where('age','>', '18'));
})
Outputs:
create view `users_view` (`first_name`) as select `first_name` from `users` where `age` > '18'
```

createViewOrReplace — `knex.schema.createViewOrReplace(tableName, callback)`

Creates a new view or replace it on the database, with a callback function to modify the view's structure, using the schema-building commands. You need to specify at least the same columns in same order (you can add extra columns). In SQLite, this function generate drop/create view queries (view columns can be different).

```
knex.schema.createViewOrReplace('users_view', function (view) {
  view.columns(['first_name']);
  view.as(knex('users').select('first_name').where('age','>', '18'));
})
Outputs:
create or replace view `users_view` (`first_name`) as select `first_name` from `users` where `age` > '18'
```

createMaterializedView — `knex.schema.createMaterializedView(viewName, callback)`

Creates a new materialized view on the database, with a callback function to modify the view's structure, using the schema-building commands. Only on PostgreSQL, CockroachDb, Redshift and Oracle.

```
knex.schema.createMaterializedView('users_view', function (view) {
  view.columns(['first_name']);
  view.as(knex('users').select('first_name').where('age','>', '18'));
})
Error:
materialized views are not supported by this dialect.
```

refreshMaterializedView — `knex.schema.refreshMaterializedView(viewName)`

Refresh materialized view on the database. Only on PostgreSQL, CockroachDb, Redshift and Oracle.

```
knex.schema.refreshMaterializedView('users_view')
Error:
materialized views are not supported by this dialect.
```

dropView — `knex.schema.dropView(viewName)`

Drop view on the database.

```
knex.schema.dropView('users_view')
Outputs:
drop view `users_view`
```

dropViewIfExists — `knex.schema.dropViewIfExists(viewName)`

Drop view on the database if exists.

```
knex.schema.dropViewIfExists('users_view')
Outputs:
drop view if exists `users_view`
```

dropMaterializedView — `knex.schema.dropMaterializedView(viewName)`

Drop materialized view on the database. Only on PostgreSQL, CockroachDb, Redshift and Oracle.

```
knex.schema.dropMaterializedView('users_view')
Error:
materialized views are not supported by this dialect.
```

dropMaterializedViewIfExists — `knex.schema.dropMaterializedViewIfExists(viewName)`

Drop materialized view on the database if exists. Only on PostgreSQL, CockroachDb, Redshift and Oracle.

```
knex.schema.dropMaterializedViewIfExists('users_view')
Error:
materialized views are not supported by this dialect.
```

renameView — `knex.schema.renameView(viewName)`

Rename a existing view in the database. Not supported by Oracle and SQLite.


```
knex.schema.renameView('users_view')
Outputs:
rename table `users_view` to `undefined`
```

alterView — `knex.schema.alterView(viewName)`

Alter view to rename columns or change default values. Only available on PostgreSQL, MSSQL and Redshift.

```
knex.schema.alterView('view_test', function (view) {
  view.column('first_name').rename('name_user');
  view.column('bio').defaultTo('empty');
})
Error:
rename column of views is not supported by this dialect.
```

generateDdlCommands — `knex.schema.generateDdlCommands()`

Generates complete SQL commands for applying described schema changes, without executing anything. Useful when knex is being used purely as a query builder. Generally produces same result as .toSQL(), with a notable exception with SQLite, which relies on asynchronous calls to the database for building part of its schema modification statements

```
const ddlCommands = knex.schema.alterTable(
  'users',
  (table) => {
    table
      .foreign('companyId')
      .references('company.companyId')
      .withKeyName('fk_fkey_company');
  }
).generateDdlCommands();
```

raw — `knex.schema.raw(statement)`

Run an arbitrary sql query in the schema builder chain.

```
knex.schema.raw("SET sql_mode='TRADITIONAL'")
  .table('users', function (table) {
    table.dropColumn('name');
    table.string('first_name');
    table.string('last_name');
  })
Outputs:
SET sql_mode='TRADITIONAL';
alter table `users` add `first_name` varchar(255), add `last_name` varchar(255);
alter table `users` drop `name`
```

queryContext — `knex.schema.queryContext(context)`

Allows configuring a context to be passed to the wrapIdentifier hook. The context can be any kind of value and will be passed to `wrapIdentifier` without modification.

```
knex.schema.queryContext({ foo: 'bar' })
  .table('users', function (table) {
    table.string('first_name');
    table.string('last_name');
  })
```

The context configured will be passed to `wrapIdentifier` for each identifier that needs to be formatted, including the table and column names. However, a different context can be set for the column names via table.queryContext.

Calling `queryContext` with no arguments will return any context configured for the schema builder instance.

dropSchema — `knex.schema.dropSchema(schemaName, [cascade])`

Drop a schema, specified by the schema's name, with optional cascade option (default to false). Only supported by PostgreSQL.

```
//drop schema 'public'
knex.schema.dropSchema('public')
//drop schema 'public' cascade
knex.schema.dropSchema('public', true)
```

dropSchemaIfExists — `knex.schema.dropSchemaIfExists(schemaName, [cascade])`

Drop a schema conditionally if the schema exists, specified by the schema's name, with optional cascade option (default to false). Only supported by PostgreSQL.

```
//drop schema if exists 'public'
knex.schema.dropSchemaIfExists('public')
//drop schema if exists 'public' cascade
knex.schema.dropSchemaIfExists('public', true)
```

Schema Building:

dropColumn — `table.dropColumn(name)`

Drops a column, specified by the column's name

dropColumns — `table.dropColumns(*columns)`

Drops multiple columns, taking a variable number of column names.

renameColumn — `table.renameColumn(from, to)`

Renames a column from one name to another.

increments — `table.increments(name, options={[primaryKey: boolean = true])`

Adds an auto incrementing column. In PostgreSQL this is a serial; in Amazon Redshift an integer identity(1,1). This will be used as the primary key for the table if the column isn't in another primary key. Also available is a bigIncrements if you wish to add a bigint incrementing number (in PostgreSQL bigserial). Note that a primary key is created by default if the column isn't in primary key (with primary function), but you can override this behaviour by passing the `primaryKey` option. If you use this function with primary function, the column is added to the composite primary key. With SQLite, autoincrement column need to be a primary key, so if primary function is used, primary keys are transformed in unique index. MySQL don't support autoincrement column without primary key, so multiple queries are generated to create int column, add increments column to composite primary key then modify the column to autoincrement column.


```
// create table 'users' with a primary key using 'increments()'
knex.schema.createTable('users', function (table) {
  table.increments('userId');
  table.string('name');
});

// create table 'users' with a composite primary key ('userId', 'name').
// increments doesn't generate primary key.
knex.schema.createTable('users', function (table) {
  table.primary(['userId', 'name']);
  table.increments('userId');
  table.string('name');
});

// reference the 'users' primary key in new table 'posts'
knex.schema.createTable('posts', function (table) {
  table.integer('author').unsigned().notNullable();
  table.string('title', 30);
  table.string('content');

  table.foreign('author').references('userId').inTable('users');
});
```

A primaryKey option may be passed, to disable to automatic primary key creation:

```
// create table 'users' with a primary key using 'increments()'
//  but also increments field 'other_id' that does not need primary key
knex.schema.createTable('users', function (table) {
  table.increments('id');
  table.increments('other_id', { primaryKey: false });
});
```

integer — `table.integer(name, length)`

Adds an integer column. On PostgreSQL you cannot adjust the length, you need to use other option such as bigInteger, etc

bigInteger — `table.bigInteger(name)`

In MySQL or PostgreSQL, adds a bigint column, otherwise adds a normal integer. Note that bigint data is returned as a string in queries because JavaScript may be unable to parse them without loss of precision.

tinyint — `table.tinyint(name, length)`

Adds a tinyint column

smallint — `table.smallint(name)`

Adds a smallint column

mediumint — `table.mediumint(name)`

Adds a mediumint column

bigint — `table.bigint(name)`

Adds a bigint column

text — `table.text(name, [textType])`

Adds a text column, with optional textType for MySql text datatype preference. textType may be mediumtext or longtext, otherwise defaults to text.

string — `table.string(name, [length])`

Adds a string column, with optional length defaulting to 255.

float — `table.float(column, [precision], [scale])`

Adds a float column, with optional precision (defaults to 8) and scale (defaults to 2).

double — `table.double(column, [precision], [scale])`

Adds a double column, with optional precision (defaults to 8) and scale (defaults to 2). In SQLite/MSSQL this is a float with no precision/scale; In PostgreSQL this is a double precision; In Oracle this is a number with matching precision/scale.

decimal — `table.decimal(column, [precision], [scale])`

Adds a decimal column, with optional precision (defaults to 8) and scale (defaults to 2). Specifying NULL as precision creates a decimal column that can store numbers of any precision and scale. (Only supported for Oracle, SQLite, Postgres)

boolean — `table.boolean(name)`

Adds a boolean column.

date — `table.date(name)`

Adds a date column.

datetime — `table.datetime(name, options={[useTz: boolean], [precision: number]})`

Adds a datetime column. By default PostgreSQL creates column with timezone (timestamptz type). This behaviour can be overriden by passing useTz option (which is by default true for PostgreSQL). MySQL and MSSQL do not have useTz option.

A precision option may be passed:

```
table.datetime('some_time', { precision: 6 }).defaultTo(knex.fn.now(6))
```

time — `table.time(name, [precision])`

Adds a time column, with optional precision for MySQL. Not supported on Amazon Redshift.

In MySQL a precision option may be passed:

```
table.time('some_time', { precision: 6 })
```

timestamp — `table.timestamp(name, options={[useTz: boolean], [precision: number]})`

Adds a timestamp column. By default PostgreSQL creates column with timezone (timestamptz type) and MSSQL does not (datetime2). This behaviour can be overriden by passing the useTz option (which is by default false for MSSQL and true for PostgreSQL). MySQL does not have useTz option.

```
table.timestamp('created_at').defaultTo(knex.fn.now());
```

In PostgreSQL and MySQL a precision option may be passed:

```
table.timestamp('created_at', { precision: 6 }).defaultTo(knex.fn.now(6));
```

In PostgreSQL and MSSQL a timezone option may be passed:

```
table.timestamp('created_at', { useTz: true });
```

timestamps — `table.timestamps([useTimestamps], [defaultToNow], [useCamelCase])`

Adds created_at and updated_at columns on the database, setting each to datetime types. When true is passed as the first argument a timestamp type is used instead. Both columns default to being not null and using the current timestamp when true is passed as the second argument. Note that on MySQL the .timestamps() only have seconds precision, to get better precision use the .datetime or .timestamp methods directly with precision. If useCamelCase is true, the name of columns are createdAt and updatedAt.

dropTimestamps — `table.dropTimestamps([useCamelCase])`

Drops the columns created_at and updated_at from the table, which can be created via timestamps. If useCamelCase is true, the name of columns are createdAt and updatedAt.

binary — `table.binary(name, [length])`

Adds a binary column, with optional length argument for MySQL.

enum / enu — `table.enu(col, values, [options])`

Adds a enum column, (aliased to enu, as enum is a reserved word in JavaScript). Implemented as unchecked varchar(255) on Amazon Redshift. Note that the second argument is an array of values. Example:

```
table.enu('column', ['value1', 'value2'])
```

For Postgres, an additional options argument can be provided to specify whether or not to use Postgres's native TYPE:

```
table.enu('column', ['value1', 'value2'], { useNative: true, enumName: 'foo_type' })
```

It will use the values provided to generate the appropriate TYPE. Example:

```
CREATE TYPE "foo_type" AS ENUM ('value1', 'value2');
```

To use an existing native type across columns, specify 'existingType' in the options (this assumes the type has already been created):

Note: Since the enum values aren't utilized for a native && existing type, the type being passed in for values is immaterial.

```
table.enu('column', null, { useNative: true, existingType: true, enumName: 'foo_type' })
```

If you want to use existing enums from a schema, different from the schema of your current table, specify 'schemaName' in the options:

```
table.enu('column', null, { useNative: true, existingType: true, enumName: 'foo_type', schemaName: 'public' })
```

Knex does not provide any way to alter enumerations after creation. To change an enumeration later on you must use Knex.raw, and the appropriate command for your database.

json — `table.json(name)`

Adds a json column, using the built-in json type in PostgreSQL, MySQL and SQLite, defaulting to a text column in older versions or in unsupported databases.

For PostgreSQL, due to incompatibility between native array and json types, when setting an array (or a value that could be an array) as the value of a json or jsonb column, you should use JSON.stringify() to convert your value to a string prior to passing it to the query builder, e.g.

```
knex.table('users')
  .where({id: 1})
  .update({json_data: JSON.stringify(mightBeAnArray)});
```

jsonb — `table.jsonb(name)`

Adds a jsonb column. Works similar to table.json(), but uses native jsonb type if possible.

uuid — `table.uuid(name, options=({[useBinaryUuid:boolean]})`

Adds a uuid column - this uses the built-in uuid type in PostgreSQL, and falling back to a char(36) in other databases by default. If useBinaryUuid is true, binary(16) is used. See uuidToBin function to convert uuid in binary before inserting and binToUuid to convert binary uuid to uuid.

geometry — `table.geometry(name)`

Adds a geometry column. Supported by SQLite, MSSQL and PostgreSQL.

```
knex.schema.createTable(tblName, (table) => {
  table.geometry('geometryColumn');
});
```

geography — `table.geography(name)`

Adds a geography column. Supported by SQLite, MSSQL and PostgreSQL (in PostGIS extension).

```
knex.schema.createTable(tblName, (table) => {
  table.geography('geographyColumn');
});
```

point — `table.point(name)`

Add a point column. Not supported by CockroachDB and MSSQL.

```
knex.schema.createTable(tblName, (table) => {
  table.point('pointColumn');
});
```

comment — `table.comment(value)`

Sets the comment for a table.

engine — `table.engine(val)`

Sets the engine for the database table, only available within a createTable call, and only applicable to MySQL.

charset — `table.charset(val)`

Sets the charset for the database table, only available within a createTable call, and only applicable to MySQL.

collate — `table.collate(val)`

Sets the collation for the database table, only available within a createTable call, and only applicable to MySQL.

inherits — `table.inherits(val)`

Sets the tables that this table inherits, only available within a createTable call, and only applicable to PostgreSQL.

specificType — `table.specificType(name, type)`

Sets a specific type for the column creation, if you'd like to add a column type that isn't supported here.

index — `table.index(columns, [indexName], options={[indexType: string], [storageEngineIndexType: 'btree'|'hash'], [predicate: QueryBuilder]})`

Adds an index to a table over the given columns. A default index name using the columns is used unless indexName is specified. In MySQL, the storage engine index type may be 'btree' or 'hash' index types, more info in Index Options section : https://dev.mysql.com/doc/refman/8.0/en/create-index.html. The indexType can be optionally specified for PostgreSQL and MySQL. Amazon Redshift does not allow creating an index. In PostgreSQL, SQLite and MSSQL a partial index can be specified by setting a 'where' predicate.

```
knex.table('users', function (table) {
  table.index(['name', 'last_name'], 'idx_name_last_name', {
    indexType: 'FULLTEXT',
    storageEngineIndexType: 'hash',
    predicate: knex.whereNotNull('email'),
  });
});
```

dropIndex — `table.dropIndex(columns, [indexName])`

Drops an index from a table. A default index name using the columns is used unless indexName is specified (in which case columns is ignored). Amazon Redshift does not allow creating an index.

setNullable — `table.setNullable(column)`

Makes table column nullable.

dropNullable — `table.dropNullable(column)`

Makes table column not nullable. Note that this operation will fail if there are already null values in this column.

primary — `table.primary(columns, options={[constraintName:string],[deferrable:'not deferrable'|'deferred'|'immediate']})`

Create a primary key constraint on table using input `columns`. If you need to create a composite primary key, pass an array of columns to `columns`. Constraint name defaults to `tablename_pkey` unless `constraintName` is specified. On Amazon Redshift, all columns included in a primary key must be not nullable. Deferrable primary constraint are supported on Postgres and Oracle and can be set by passing deferrable option to options object.

```
knex.schema.alterTable('users', function(t) {
  t.unique('email')
})
knex.schema.alterTable('job', function(t) {
  t.primary('email',{constraintName:'users_primary_key',deferrable:'deferred'})
})
```

Note: If you want to chain primary() while creating new column you can use primary

unique — `table.unique(columns, options={[indexName: string], [deferrable:'not deferrable'|'immediate'|'deferred'], [storageEngineIndexType:'btree'|'hash'], [useConstraint:true|false]})`

Adds an unique index to a table over the given `columns`. In MySQL, the storage engine index type may be 'btree' or 'hash' index types, more info  Index options GitHub section : https://dev.mysql.com/doc/refman/8.0/en/create-index.html. A default index name using the columns is used unless indexName is specified. If you need to create a composite index, pass an array of column to `columns`. Deferrable unique constraint are supported on Postgres and Oracle and can be set by passing deferrable option to options object. In MSSQL you can set the `useConstraint` option to true to create a unique constraint instead of a unique index.

```
knex.schema.alterTable('users', function(t) {
  t.unique('email')
})
knex.schema.alterTable('job', function(t) {
  t.unique(['account_id', 'program_id'], {indexName: 'users_composite_index', deferrable:'deferred', storageEngineIndexType: 'hash'})
})
knex.schema.alterTable('job', function(t) {
  t.unique(['account_id', 'program_id'], {indexName: 'users_composite_index', useConstraint:true})
})
```

Note: If you want to chain unique() while creating new column you can use unique

foreign — `table.foreign(columns, [foreignKeyName])[.onDelete(statement).onUpdate(statement).withKeyName(foreignKeyName).deferrable(type)]`

Adds a foreign key constraint to a table for an existing column using `table.foreign(column).references(column)` or multiple columns using `table.foreign(columns).references(columns).inTable(table)`.

A default key name using the columns is used unless `foreignKeyName` is specified.

You can also chain `onDelete()` and/or `onUpdate()` to set the reference option `(RESTRICT, CASCADE, SET NULL, NO ACTION)` for the operation. You can also chain `withKeyName()` to override default key name that is generated from table and column names (result is identical to specifying second parameter to function `foreign()`).

Deferrable foreign constraint is supported on Postgres and Oracle and can be set by chaining `.deferrable(type)`

Note that using `foreign()` is the same as `column.references(column)` but it works for existing columns.

```
knex.schema.table('users', function (table) {
  table.integer('user_id').unsigned()
  table.foreign('user_id').references('Items.user_id_in_items').deferrable('deferred')
})
```

dropForeign — `table.dropForeign(columns, [foreignKeyName])`

Drops a foreign key constraint from a table. A default foreign key name using the columns is used unless foreignKeyName is specified (in which case columns is ignored).

dropUnique — `table.dropUnique(columns, [indexName])`

Drops a unique key constraint from a table. A default unique key name using the columns is used unless indexName is specified (in which case columns is ignored).

dropPrimary — `table.dropPrimary([constraintName])`

Drops the primary key constraint on a table. Defaults to tablename_pkey unless constraintName is specified.

queryContext — `table.queryContext(context)`

Allows configuring a context to be passed to the wrapIdentifier hook for formatting table builder identifiers. The context can be any kind of value and will be passed to `wrapIdentifier` without modification.

```
knex.schema.table('users', function (table) {
  table.queryContext({ foo: 'bar' });
  table.string('first_name');
  table.string('last_name');
})
```

This method also enables overwriting the context configured for a schema builder instance via schema.queryContext:

```
knex.schema.queryContext('schema context')
  .table('users', function (table) {
    table.queryContext('table context');
    table.string('first_name');
    table.string('last_name');
})
```

Note that it's also possible to overwrite the table builder context for any column in the table definition:

```
knex.schema.queryContext('schema context')
  .table('users', function (table) {
    table.queryContext('table context');
    table.string('first_name').queryContext('first_name context');
    table.string('last_name').queryContext('last_name context');
})
```

Calling `queryContext` with no arguments will return any context configured for the table builder instance.

Chainable Methods:

The following three methods may be chained on the schema building methods, as modifiers to the column.

alter — `column.alter(options={[alterNullable: boolean = true, alterType: boolean = true])`

Marks the column as an alter / modify, instead of the default add. Note: This only works in .alterTable() and is not supported by SQlite or Amazon Redshift. Alter is *not* done incrementally over older column type so if you like to add `notNullable` and keep the old default value, the alter statement must contain both `.notNullable().defaultTo(1).alter()`. If one just tries to add `.notNullable().alter()` the old default value will be dropped. Nullable alterations are done only if alterNullable is true. Type alterations are done only if alterType is true.

```
knex.schema.alterTable('user', function(t) {
  t.increments().primary(); // add
  // drops previous default value from column, change type to string and add not nullable constraint
  t.string('username', 35).notNullable().alter();
```

```
  // drops both not null constraint and the default value
  t.integer('age').alter();
  // if alterNullable is false, drops only the default value
  t.integer('age').alter({alterNullable : false});
  // if alterType is false, type of column is not altered.
  t.integer('age').alter({alterType : false});
});
```

index — `column.index([indexName], options=({[indexType: string], [storageEngineIndexType: 'btree'|'hash'], [predicate: QueryBuilder]}))`

Specifies a field as an index. If an indexName is specified, it is used in place of the standard index naming convention of tableName_columnName. In MySQL, the storage engine index type may be 'btree' or 'hash' index types, more info in Index Options section : https://dev.mysql.com/doc/refman/8.0/en/create-index.html. The indexType can be optionally specified for PostgreSQL and MySQL. No-op if this is chained off of a field that cannot be indexed. In PostgreSQL, SQLite and MSSQL a partial index can be specified by setting a 'where' predicate.

primary — `column.primary(options=({[constraintName:string],[deferrable:'not deferrable'|'deferred'|'immediate']}));`

Sets a primary key constraint on `column`. Constraint name defaults to `tablename_pkey` unless `constraintName` is specified. On Amazon Redshift, all columns included in a primary key must be not nullable. Deferrable primary constraint are supported on Postgres and Oracle and can be set by passing deferrable option to options object.

```
knex.schema.table('users', function (table) {
  table.integer('user_id').primary('email',{constraintName:'users_primary_key',deferrable:'deferred'})
})
```

Note: If you want to create primary constraint on existing column use primary

unique — `column.unique(options={[indexName:string],[deferrable:'not deferrable'|'immediate'|'deferred']})`

Sets the `column` as unique. On Amazon Redshift, this constraint is not enforced, but it is used by the query planner. Deferrable unqiue constraint are supported on Postgres and Oracle and can be set by passing deferrable option to options object.

```
knex.schema.table('users', function (table) {
  table.integer('user_id').unique({indexName:'user_unqiue_id', deferrable:'immediate'})
})
```

Note: If you want to create unique constraint on existing column use unique

references — `column.references(column)`

Sets the "column" that the current column references as a foreign key. "column" can either be "." syntax, or just the column name followed up with a call to inTable to specify the table.

inTable — `column.inTable(table)`

Sets the "table" where the foreign key column is located after calling column.references.

onDelete — `column.onDelete(command)`

Sets the SQL command to be run "onDelete".

onUpdate — `column.onUpdate(command)`

Sets the SQL command to be run "onUpdate".

defaultTo — `column.defaultTo(value, options={[constraintName: string = undefined]))`

Sets the default value for the column on an insert.

In MSSQL a constraintName option may be passed to ensure a specific constraint name:

```
column.defaultTo('value', { constraintName: 'df_table_value' });
```

unsigned — `column.unsigned()`

Specifies an integer as unsigned. No-op if this is chained off of a non-integer field.

notNullable — `column.notNullable()`

Adds a not null on the current column being created.

nullable — `column.nullable()`

Default on column creation, this explicitly sets a field to be nullable.

first — `column.first()`

Sets the column to be inserted on the first position, only used in MySQL alter tables.

after — `column.after(field)`

Sets the column to be inserted after another, only used in MySQL alter tables.

comment — `column.comment(value)`

Sets the comment for a column.

```
knex.schema.createTable('accounts', function(t) {
  t.increments().primary();
  t.string('email').unique().comment('This is the email field');
});
```

collate — `column.collate(collation)`

Sets the collation for a column (only works in MySQL). Here is a list of all available collations: https://dev.mysql.com/doc/refman/5.5/en/charset-charsets.html

```
knex.schema.createTable('users', function(t) {
  t.increments();
  t.string('email').unique().collate('utf8_unicode_ci');
});
```

View:

columns — `view.columns([columnNames])`

Specify the columns of the view.

```
knex.schema.createView('users_view', function (view) {
  view.columns(['first_name', 'last_name']);
  view.as(knex('users').select('first_name').where('age','>', '18'));
});
```

as — `view.as(selectQuery)`

Specify the select query of the view.

checkOption — `view.checkOption()`

Add check option on the view definition. On OracleDb, MySQL, PostgreSQL and Redshift.

localCheckOption — `view.localCheckOption()`

Add local check option on the view definition. On MySQL, PostgreSQL and Redshift.

cascadedCheckOption — `view.cascadedCheckOption()`

Add cascaded check option on the view definition. On MySQL, PostgreSQL and Redshift.

Checks:

check — `table.check(checkPredicate, [bindings], [constraintName]))`

Specify a check on table or column with raw predicate.

```
knex.schema.createTable('product', function (table) {
  table.integer('price_min');
  table.integer('price');
  table.check('?? >= ??', ['price', 'price_min']);
})
```

checkPositive — `column.checkPositive([constraintName])`

Specify a check on column that test if the value of column is positive.

```
knex.schema.createTable('product', function (table) {
  table.integer('price').checkPositive();
})
```

checkNegative — `column.checkNegative([constraintName])`

Specify a check on column that test if the value of column is negative.

```
knex.schema.createTable('product', function (table) {
  table.integer('price_decrease').checkNegative();
})
```

checkIn — `column.checkIn(values, [constraintName])`

Specify a check on column that test if the value of column is contained in a set of specified values.

```
knex.schema.createTable('product', function (table) {
  table.string('type').checkIn(['table', 'chair', 'sofa']);
})
```

checkNotIn — `column.checkNotIn(values, [constraintName])`

Specify a check on column that test if the value of column is not contains in a set of specified values.

```
knex.schema.createTable('product', function (table) {
  table.string('type').checkNotIn(['boot', 'shoe']);
})
```

checkBetween — `column.checkBetween(values, [constraintName])`

Specify a check on column that test if the value of column is within a range of values.

```
knex.schema.createTable('product', function (table) {
  table.integer('price').checkBetween([0, 100]);
})
// You can add checks on multiple intervals
knex.schema.createTable('product', function (table) {
  table.integer('price').checkBetween([ [0, 20], [30,40] ]);
})
```

checkLength — `column.checkLength(operator, length, [constraintName])`

Specify a check on column that test if the length of a string match the predicate.

```
knex.schema.createTable('product', function (table) {
  // operator can be =, !=, <=, >=, <, >
```

```
  t.varchar('phone').checkLength('=', 8);
})
```

checkRegex — `column.checkRegex(regex, [constraintName])`

Specify a check on column that test if the value match the specified regular expression. In MSSQL only simple pattern matching in supported but not regex syntax.

```
knex.schema.createTable('product', function (table) {
  table.string('phone').checkRegex('[0-9]{8}');
  // In MSSQL, {8} syntax don't work, you need to duplicate [0-9].
  table.string('phone').checkRegex('[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]');
})
```

dropChecks — `table.dropChecks([checkConstraintNames])`

Drop checks constraint given an array of constraint names.

```
knex.schema.createTable('product', function (table) {
  table.integer('price').checkPositive('price_check'):
  table.integer('price_proportion').checkBetween([0, 100],'price_proportion_check'):
  table.dropChecks(['price_check', 'price_proportion_check']);
})
```

Raw

Sometimes you may need to use a raw expression in a query. Raw query object may be injected pretty much anywhere you want, and using proper bindings can ensure your values are escaped properly, preventing SQL-injection attacks.

Raw Parameter Binding:

One can paramterize sql given to `knex.raw(sql, bindings)`. Parameters can be positional named. One can also choose if parameter should be treated as value or as sql identifier e.g. in case of `'TableName.ColumnName'` reference.

```
knex('users')
  .select(knex.raw('count(*) as user_count, status'))
  .where(knex.raw(1))
  .orWhere(knex.raw('status <> ?', [1]))
  .groupBy('status')
Outputs:
select count(*) as user_count, status from `users` where 1 or status <> 1 group by `status`
```

Positional bindings `?` are interpreted as values and `??` are interpreted as identifiers.

```
knex('users').where(knex.raw('?? = ?', ['user.name', 1]))
Outputs:
select * from `users` where `user`.`name` = 1
```

Named bindings such as `:name` are interpreted as values and `:name:` interpreted as identifiers. Named bindings are processed so long as the value is anything other than `undefined`.

```
knex('users')
  .where(knex.raw(':name: = :thisGuy or :name: = :otherGuy or :name: = :undefinedBinding', {
    name: 'users.name',
    thisGuy: 'Bob',
    otherGuy: 'Jay',
    undefinedBinding: undefined
  }))
Error:
Undefined binding(s) detected for keys [undefinedBinding] when compiling RAW query: `users`.`name` = ? or `users`.`name` = ? or `users`.`name` = :unde
```

For simpler queries where one only has a single binding, `.raw` can accept said binding as its second parameter.

```
knex('users')
  .where(
    knex.raw('LOWER("login") = ?', 'knex')
  )
  .orWhere(
    knex.raw('accesslevel = ?', 1)
  )
  .orWhere(
    knex.raw('updtime = ?', '01-01-2016')
  )
Outputs:
select * from `users` where LOWER("login") = 'knex' or accesslevel = 1 or updtime = '01-01-2016'
```

Since there is no unified syntax for array bindings, instead you need to treat them as multiple values by adding `?` directly in your query.

```
const myArray = [1,2,3]
knex.raw('select * from users where id in (' + myArray.map(_ => '?').join(',') + ')', [...myArray]);
// query will become: select * from users where id in (?, ?, ?) with bindings [1,2,3]
```

To prevent replacement of `?` one can use the escape sequence `\\?`.

```
knex.select('*').from('users').where('id', '=', 1).whereRaw('?? \\? ?', ['jsonColumn', 'jsonKey'])
Outputs:
select * from `users` where `id` = 1 and `jsonColumn` ? 'jsonKey'
```

To prevent replacement of named bindings one can use the escape sequence `\\:`.

```
knex.select('*').from('users').whereRaw(":property: = '\\:value' OR \\:property: = :value", {
  property: 'name',
  value: 'Bob'
})
Outputs:
select * from `users` where `name` = ':value' OR :property: = 'Bob'
```

Raw Expressions:

Raw expressions are created by using `knex.raw(sql, [bindings])` and passing this as a value for any value in the query chain.

```
knex('users')
  .select(knex.raw('count(*) as user_count, status'))
  .where(knex.raw(1))
  .orWhere(knex.raw('status <> ?', [1]))
  .groupBy('status')
Outputs:
select count(*) as user_count, status from `users` where 1 or status <> 1 group by `status`
```

Raw Queries:

The `knex.raw` may also be used to build a full query and execute it, as a standard query builder query would be executed. The benefit of this is that it uses the connection pool and provides a standard interface for the different client libraries.

```
knex.raw('select * from users where id = ?', [1]).then(function(resp) { ... });
```

Note that the response will be whatever the underlying sql library would typically return on a normal query, so you may need to look at the documentation for the base library the queries are executing against to determine how to handle the response.

Wrapped Queries:

The raw query builder also comes with a `wrap` method, which allows wrapping the query in a value:

```
const subcolumn = knex.raw('select avg(salary) from employee where dept_no = e.dept_no')
.wrap('(', ') avg_sal_dept');

knex.select('e.lastname', 'e.salary', subcolumn)
.from('employee as e')
.whereRaw('dept_no = e.dept_no')
Outputs:
select `e`.`lastname`, `e`.`salary`, (select avg(salary) from employee where dept_no = e.dept_no) avg_sal_dept from `employee` as `e` where dept_no =
```

Note that the example above be achieved more easily using the as method.

```
const subcolumn = knex.avg('salary')
.from('employee')
.whereRaw('dept_no = e.dept_no')
.as('avg_sal_dept');

knex.select('e.lastname', 'e.salary', subcolumn)
.from('employee as e')
.whereRaw('dept_no = e.dept_no')
Outputs:
select `e`.`lastname`, `e`.`salary`, (select avg(`salary`) from `employee` where dept_no = e.dept_no) as `avg_sal_dept` from `employee` as `e` where
```

Ref

Can be used to create references in a query, such as column- or tablenames. This is a good and shorter alternative to using `knex.raw('??', 'tableName.columName')` which essentially does the same thing.

Usage:

`knex.ref` can be used essentially anywhere in a build-chain. Here is an example:

```
knex(knex.ref('Users').withSchema('TenantId'))
  .where(knex.ref('Id'), 1)
  .orWhere(knex.ref('Name'), 'Admin')
  .select(['Id', knex.ref('Name').as('Username')])
Outputs:
select `Id`, `Name` as `Username` from `TenantId`.`Users` where `Id` = 1 or `Name` = 'Admin'
```

withSchema:

The Ref function supports schema using `.withSchema(string)`:

```
knex(knex.ref('users').withSchema('TenantId')).select()
Outputs:
select * from `TenantId`.`users`
```

alias:

Alias is supported using `.alias(string)`

```
knex('users')
  .select(knex.ref('Id').as('UserId'))
Outputs:
select `Id` as `UserId` from `users`
```

Utility

A collection of utilities that the knex library provides for convenience.

Batch Insert

The `batchInsert` utility will insert a batch of rows wrapped inside a transaction *(which is automatically created unless explicitly given a transaction using transacting)*, at a given `chunkSize`.

It's primarily designed to be used when you have thousands of rows to insert into a table.

By default, the `chunkSize` is set to 1000.

BatchInsert also allows for returning values and supplying transactions using transacting.


```
const rows = [{...}, {...}];
const chunkSize = 30;
knex.batchInsert('TableName', rows, chunkSize)
  .returning('id')
  .then(function(ids) { ... })
  .catch(function(error) { ... });

knex.transaction(function(tr) {
  return knex.batchInsert('TableName', rows, chunkSize)
    .transacting(tr)
  })
  .then(function() { ... })
  .catch(function(error) { ... });
```

now — `knex.fn.now(precision)`

Return the current timestamp with a precision (optional)

```
table.datetime('some_time', { precision: 6 }).defaultTo(knex.fn.now(6))
```

binToUuid — `knex.fn.binToUuid(binaryUuid)`

Convert a binary uuid (binary(16)) to a string uuid (char(36))

```
knex.schema.createTable('uuid_table', (t) => {
  t.uuid('uuid_col_binary', { useBinaryUuid: true });
});
knex('uuid_table').insert({
  uuid_col_binary:  knex.fn.uuidToBin('3f06af63-a93c-11e4-9797-00505690773f'),
});
```

uuidToBin — `knex.fn.uuidToBin(uuid)`

Convert a uuid (char(16)) to a binary uuid (binary(36))

```
const res = await knex('uuid_table').select('uuid_col_binary');
knex.fn.binToUuid(res[0].uuid_col_binary)
```

Interfaces

Knex.js provides several options to deal with query output. The following methods are present on the query builder, schema builder, and the raw builder:

Promises

Promises are the preferred way of dealing with queries in knex, as they allow you to return values from a fulfillment handler, which in turn become the value of the promise. The main benefit of promises are the ability to catch thrown errors without crashing the node app, making your code behave like a **.try / .catch / .finally** in synchronous code.

```
knex.select('name')
  .from('users')
  .where('id', '>', 20)
  .andWhere('id', '<', 200)
  .limit(10)
  .offset(x)
  .then(function(rows) {
    return _.pluck(rows, 'name');
  })
  .then(function(names) {
    return knex.select('id').from('nicknames').whereIn('nickname', names);
  })
  .then(function(rows) {
    console.log(rows);
  })
  .catch(function(error) {
    console.error(error)
  });
```

then — `.then(onFulfilled, [onRejected])`

Coerces the current query builder chain into a promise state, accepting the resolve and reject handlers as specified by the Promises/A+ spec. As stated in the spec, more than one call to the then method for the current query chain will resolve with the same value, in the order they were called; the query will not be executed multiple times.

```
knex.select('*')
  .from('users')
  .where({name: 'Tim'})
  .then(function(rows) {
    return knex.insert({user_id: rows[0].id, name: 'Test'}, 'id').into('accounts');
  })
  .then(function(id) {
    console.log('Inserted Account ' + id);
  })
  .catch(function(error) { console.error(error); });
```

catch — `.catch(onRejected)`

Coerces the current query builder into a promise state, catching any error thrown by the query, the same as calling .then(null, onRejected).

```
return knex.insert({id: 1, name: 'Test'}, 'id')
  .into('accounts')
  .catch(function(error) {
    console.error(error);
  }).then(function() {
    return knex.select('*')
      .from('accounts')
      .where('id', 1);
```

```
}).then(function(rows) {
  console.log(rows[0]);
})
.catch(function(error) {
  console.error(error);
});
```

Callbacks

asCallback — `.asCallback(callback)`

If you'd prefer a callback interface over promises, the asCallback function accepts a standard node style callback for executing the query chain. Note that as with the then method, subsequent calls to the same query chain will return the same result.

```
knex.select('name').from('users')
  .where('id', '>', 20)
  .andWhere('id', '<', 200)
  .limit(10)
  .offset(x)
  .asCallback(function(err, rows) {
    if (err) return console.error(err);
    knex.select('id').from('nicknames')
      .whereIn('nickname', _.pluck(rows, 'name'))
      .asCallback(function(err, rows) {
        if (err) return console.error(err);
        console.log(rows);
      });
  });
```

Streams

Streams are a powerful way of piping data through as it comes in, rather than all at once. You can read more about streams here at substack's stream handbook. See the following for example uses of stream & pipe. If you wish to use streams with PostgreSQL, you must also install the pg-query-stream module. If you wish to use streams with the `pgnative` dialect, please be aware that the results will not be streamed as they are received, but rather streamed after the entire result set has returned. On an HTTP server, make sure to manually close your streams if a request is aborted.

stream — `.stream([options], [callback])`

If called with a callback, the callback is passed the stream and a promise is returned. Otherwise, the readable stream is returned.

```
// Retrieve the stream:
const stream = knex.select('*').from('users').stream();
stream.pipe(writableStream);

// With options:
const stream = knex.select('*').from('users').stream({highWaterMark: 5});
stream.pipe(writableStream);

// Use as a promise:
const stream = knex.select('*').from('users')
  .where(knex.raw('id = ?', [1]))
  .stream(function(stream) {
    stream.pipe(writableStream);
  })
  .then(function() { // ... })
  .catch(function(e) { console.error(e); });
```

pipe — `.pipe(writableStream)`

Pipe a stream for the current query to a writableStream.

```
const stream = knex.select('*').from('users').pipe(writableStream);
```

Events

query

A query event is fired just before a query takes place, providing data about the query, including the connection's `__knexUid` / `__knexTxId` properties and any other information about the query as described in toSQL. Useful for logging all queries throughout your application.

```
knex.select('*')
  .from('users')
  .on('query', function(data) {
    app.log(data);
  })
  .then(function() {
    // ...
  });
```

query-error

A query-error event is fired when an error occurs when running a query, providing the error object and data about the query, including the connection's `__knexUid` / `__knexTxId` properties and any other information about the query as described in toSQL. Useful for logging all query errors throughout your application.

```
knex.select(['NonExistentColumn'])
  .from('users')
  .on('query-error', function(error, obj) {
    app.log(error);
  })
  .then(function() { // ... })
  .catch(function(error) {
    // Same error object as the query-error event provides.
  });
```

query-response

A query-response event is fired when a successful query has been run, providing the response of the query and data about the query, including the connection's `__knexUid` / `__knexTxId` properties and any other information about the query as described in toSQL, and finally the query builder used for the query.


```
knex.select('*')
  .from('users')
  .on('query-response', function(response, obj, builder) {
    // ...
  })
  .then(function(response) {
    // Same response as the emitted event
  })
  .catch(function(error) { });
```

start

A `start` event is fired right before a query-builder is compiled. Note: While this event can be used to alter a builders state prior to compilation it is not to be recommended. Future goals include ways of doing this in a different manner such as hooks.

```
knex.select('*')
  .from('users')
  .on('start', function(builder) {
    builder
    .where('IsPrivate', 0)
  })
  .then(function(Rows) {
    //Only contains Rows where IsPrivate = 0
  })
  .catch(function(error) { });
```

Other

toString — `.toString()`

Returns an array of query strings filled out with the correct values based on bindings, etc. Useful for debugging, but should not be used to create queries for running them against DB.

```
const toStringQuery = knex.select('*').from('users').where('id', 1).toString();

// Outputs: console.log(toStringQuery);
// select * from "users" where "id" = 1
```

toSQL — `.toSQL() and toSQL().toNative()`

Returns an array of query strings filled out with the correct values based on bindings, etc. Useful for debugging and building queries for running them manually with DB driver. `.toSQL().toNative()` outputs object with sql string and bindings in a dialects format in the same way that knex internally sends them to underlying DB driver.

```
knex.select('*').from('users')
  .where(knex.raw('id = ?', [1]))
  .toSQL()
// Outputs:
// {
//   bindings: [1],
//   method: 'select',
//   sql: 'select * from "users" where id = ?',
//   options: undefined,
//   toNative: function () {}
// }

knex.select('*').from('users')
  .where(knex.raw('id = ?', [1]))
  .toSQL().toNative()
// Outputs for postgresql dialect:
// {
//   bindings: [1],
//   sql: 'select * from "users" where id = $1',
// }
```

Migrations

Migrations allow for you to define sets of schema changes so upgrading a database is a breeze.

Migration CLI

The migration CLI is bundled with the knex install, and is driven by the node-liftoff module. To install globally, run:

```
$ npm install knex -g
```

The migration CLI accepts the following general command-line options. You can view help text and additional options for each command using `--help`. E.g. `knex migrate:latest --help`.

- `--debug`: Run with debugging
- `--knexfile [path]`: Specify the knexfile path
- `--knexpath [path]`: Specify the path to the knex instance
- `--cwd [path]`: Specify the working directory
- `--client [name]`: Set the DB client without a knexfile
- `--connection [address]`: Set the DB connection without a knexfile
- `--migrations-table-name`: Set the migration table name without a knexfile
- `--migrations-directory`: Set the migrations directory without a knexfile
- `--env`: environment, default: process.env.NODE_ENV || development
- `--esm`: Enables ESM module interoperability
- `--help`: Display help text for a particular command and exit.

Migrations use a **knexfile**, which specify various configuration settings for the module. To create a new knexfile, run the following:

```
$ knex init

# or for .ts

$ knex init -x ts
```

will create a sample knexfile.js - the file which contains our various database configurations. Once you have a knexfile.js, you can use the migration tool to create migration files to the specified directory (default migrations). Creating new migration files can be achieved by running:

```
$ knex migrate:make migration_name

# or for .ts

$ knex migrate:make migration_name -x ts
```

- you can also create your migration using a specific stub file, this serves as a migration template to speed up development for common migration operations
- if the --stub option is not passed, the CLI will use either the knex default stub for the chosen extension, or the config.stub file

```
$ knex migrate:make --stub

# or

$ knex migrate:make --stub
```

- if a stub path is provided, it must be relative to the knexfile.[js, ts, etc] location
- if a is used, the stub is selected by its file name. The CLI will look for this file in the config.migrations.directory folder. If the config.migrations.directory is not defined, this operation will fail

Once you have finished writing the migrations, you can update the database matching your `NODE_ENV` by running:

```
$ knex migrate:latest
```

You can also pass the `--env` flag or set `NODE_ENV` to select an alternative environment:

```
$ knex migrate:latest --env production

# or

$ NODE_ENV=production knex migrate:latest
```

To rollback the last batch of migrations:

```
$ knex migrate:rollback
```

To rollback all the completed migrations:

```
$ knex migrate:rollback --all
```

To run the next migration that has not yet been run

```
$ knex migrate:up
```

To run the specified migration that has not yet been run

```
$ knex migrate:up 001_migration_name.js
```

To undo the last migration that was run

```
$ knex migrate:down
```

To undo the specified migration that was run

```
$ knex migrate:down 001_migration_name.js
```

To list both completed and pending migrations:

```
$ knex migrate:list
```

Seed files

Seed files allow you to populate your database with test or seed data independent of your migration files.

Seed CLI

To create a seed file, run:

```
$ knex seed:make seed_name
```

Seed files are created in the directory specified in your knexfile.js for the current environment. A sample seed configuration looks like:

```
development: {
  client: ...,
  connection: { ... },
  seeds: {
      directory: './seeds/dev'
  }
}
```

If no `seeds.directory` is defined, files are created in `./seeds`. Note that the seed directory needs to be a relative path. Absolute paths are not supported (nor is it good practice).

To run seed files, execute:

```
$ knex seed:run
```

Seed files are executed in alphabetical order. Unlike migrations, *every* seed file will be executed when you run the command. You should design your seed files to reset tables as needed before inserting data.

To run specific seed files, execute:

```
$ knex seed:run --specific=seed-filename.js --specific=another-seed-filename.js
```

knexfile.js

A knexfile.js generally contains all of the configuration for your database. It can optionally provide different configuration for different environments. You may pass a `--knexfile` option to any of the command line statements to specify an alternate path to your knexfile.

Basic configuration:

```
module.exports = {
  client: 'pg',
  connection: process.env.DATABASE_URL || { user: 'me', database: 'my_app' }
};
```

you can also export an async function from the knexfile. This is useful when you need to fetch credentials from a secure location like vault

```
async function fetchConfiguration() {
  // TODO: implement me
  return {
    client: 'pg',
    connection: { user: 'me', password: 'my_pass' }
  }
}

module.exports = async () => {
  const configuration = await fetchConfiguration();
  return {
    ...configuration,
    migrations: {}
  }
};
```

Environment configuration:

```
module.exports = {
  development: {
    client: 'pg',
    connection: { user: 'me', database: 'my_app' }
  },
  production: { client: 'pg', connection: process.env.DATABASE_URL }
};
```

Custom migration:

You may provide a custom migration stub to be used in place of the default option.

```
module.exports = {
  client: 'pg',
  migrations: {
    stub: 'migration.stub'
  }
};
```

Generated migration extension:

You can control extension of generated migrations.

```
module.exports = {
  client: 'pg',
  migrations: {
    extension: 'ts'
  }
};
```

Knexfile in other languages

Knex uses Liftoff to support knexfile written in other compile-to-js languages.

Depending on the language, this may require you to install additional dependencies. The complete list of dependencies for each supported language can be found here.

Most common cases are typescript (for which typescript and ts-node packages are recommended), and coffeescript (for which coffeescript dependency is required).

If you don't specify the extension explicitly, the extension of generated migrations/seed files will be inferred from the knexfile extension

Migration API

`knex.migrate` is the class utilized by the knex migrations cli.

Each method takes an optional `config` object, which may specify the following properties:

- `directory`: a relative path to the directory containing the migration files. Can be an array of paths (default `./migrations`)
- `extension`: the file extension used for the generated migration files (default `js`)
- `tableName`: the table name used for storing the migration state (default `knex_migrations`)
- `schemaName`: the schema name used for storing the table with migration state (optional parameter, only works on DBs that support multiple schemas in a single DB, such as PostgreSQL)
- `disableTransactions`: don't run migrations inside transactions (default `false`)

- `disableMigrationsListValidation`: do not validate that all the already executed migrations are still present in migration directories (default `false`)
- `sortDirsSeparately`: if true and multiple directories are specified, all migrations from a single directory will be executed before executing migrations in the next folder (default `false`)
- `loadExtensions`: array of file extensions which knex will treat as migrations. For example, if you have typescript transpiled into javascript in the same folder, you want to execute only javascript migrations. In this case, set `loadExtensions` to `['.js']` (Notice the dot!) (default `['.co', '.coffee', '.eg', '.iced', '.js', '.litcoffee', '.ls', '.ts']`)
- `migrationSource`: specify a custom migration source, see Custom Migration Source for more info (default filesystem)

Transactions in migrations

By default, each migration is run inside a transaction. Whenever needed, one can disable transactions for all migrations via the common migration config option `config.disableTransactions` or per-migration, via exposing a boolean property `config.transaction` from a migration file:

```
exports.up = function(knex) {
  return knex.schema
    .createTable('users', function (table) {
       table.increments('id');
       table.string('first_name', 255).notNullable();
       table.string('last_name', 255).notNullable();
    })
    .createTable('products', function (table) {
       table.increments('id');
       table.decimal('price').notNullable();
       table.string('name', 1000).notNullable();
    });
};

exports.down = function(knex) {
  return knex.schema
      .dropTable("products")
      .dropTable("users");
};

exports.config = { transaction: false };
```

The same config property can be used for enabling transaction per-migration in case the common configuration has `disableTransactions: true`.

make — `knex.migrate.make(name, [config])`

Creates a new migration, with the name of the migration being added.

latest — `knex.migrate.latest([config])`

Runs all migrations that have not yet been run.

If you need to run something only after all migrations have finished their execution, you can do something like this:

```
knex.migrate.latest()
  .then(function() {
    return knex.seed.run();
  })
  .then(function() {
    // migrations are finished
  });
```

rollback — `knex.migrate.rollback([config], [all])`

Rolls back the latest migration group. If the `all` parameter is truthy, all applied migrations will be rolled back instead of just the last batch. The default value for this parameter is `false`.

up — `knex.migrate.up([config])`

Runs the specified (by `config.name` parameter) or the next chronological migration that has not yet be run.

down — `knex.migrate.down([config])`

Will undo the specified (by `config.name` parameter) or the last migration that was run.

currentVersion — `knex.migrate.currentVersion([config])`

Retrieves and returns the current migration version, as a promise. If there aren't any migrations run yet, returns "none" as the value for the currentVersion.

list — `knex.migrate.list([config])`

Will return list of completed and pending migrations

unlock — `knex.migrate.forceFreeMigrationsLock([config])`

Forcibly unlocks the migrations lock table, and ensures that there is only one row in it.

Notes about locks

A lock system is there to prevent multiple processes from running the same migration batch in the same time. When a batch of migrations is about to be run, the migration system first tries to get a lock using a `SELECT ... FOR UPDATE` statement (preventing race conditions from happening). If it can get a lock, the migration batch will run. If it can't, it will wait until the lock is released.

Please note that if your process unfortunately crashes, the lock will have to be *manually* removed with `knex migrate:unlock` in order to let migrations run again.

The locks are saved in a table called "`tableName`_lock"; it has a column called `is_locked` that `knex migrate:unlock` sets to `0` in order to release the lock. The `index` column in the lock table exists for compatibility with some database clusters that require a primary key, but is otherwise unused. There must be only one row in this table, or an error will be thrown when running migrations: "Migration table is already locked". Run `knex migrate:unlock` to ensure that there is only one row in the table.

Custom migration sources

Knex supports custom migration sources, allowing you full control of where your migrations come from. This can be useful for custom folder structures, when bundling with webpack/browserify and other scenarios.

```
// Create a custom migration source class
class MyMigrationSource {
  // Must return a Promise containing a list of migrations.
  // Migrations can be whatever you want, they will be passed as
  // arguments to getMigrationName and getMigration
  getMigrations() {
    // In this example we are just returning migration names
    return Promise.resolve(['migration1'])
  }

  getMigrationName(migration) {
    return migration;
  }

  getMigration(migration) {
    switch(migration) {
      case 'migration1':
        return {
          up(knex)   { /* ... */ }
          down(knex) { /* ... */ }
        }
    }
  }
}

// pass an instance of your migration source as knex config
knex.migrate.latest({ migrationSource: new MyMigrationSource() })
```

Webpack migration source example

An example of how to create a migration source where migrations are included in a webpack bundle.

```
const path = require('path')

class WebpackMigrationSource {
  constructor(migrationContext) {
    this.migrationContext = migrationContext
  }

  getMigrations() {
    return Promise.resolve(this.migrationContext.keys().sort())
  }

  getMigrationName(migration) {
    return path.parse(migration).base
  }

  getMigration(migration) {
    return this.migrationContext(migration)
  }
}

// pass an instance of your migration source as knex config
knex.migrate.latest({
  migrationSource: new WebpackMigrationSource(require.context('./migrations', false, /.js$/))
})

// with webpack >=5, require.context will add both the relative and absolute paths to the context
// to avoid duplicate migration errors, you'll need to filter out one or the other
// this example filters out absolute paths, leaving only the relative ones(./migrations/*.js):
knex.migrate.latest({
  migrationSource: new WebpackMigrationSource(require.context('./migrations', false, /^\.\/.*\.js$/))
})
```

ECMAScript modules (ESM) Interoperability

ECMAScript Module support for knex CLI's configuration, migration and seeds
enabled by the `--esm` flag, ECMAScript Interoperability is provided by the *'esm'* module.
You can find here more information about 'esm' superpowers.

Node 'mjs' files are handled by NodeJS own import mechanics
and do not require the use of the '--esm' flag.
But you might need it anyway for Node v10 under certain scenarios.
You can find details about NodeJS ECMAScript modules here

While it is possible to mix and match different module formats (extensions)
between your knexfile, seeds and migrations,
some format combinations will require specific NodeJS versions,
Notably mjs/cjs files will follow NodeJS import and require restrictions.
You can see here many possible scenarios,
and here some sample configurations

Node v10.* require the use of the '--experimental-module' flag in order to use the 'mjs' or 'cjs' extension.

```
# launching knex on Node v10 to use mjs/cjs modules
node --experimental-modules ./node_modules/.bin/knex $@
```

When using migration and seed files with '.cjs' or '.mjs' extensions, you will need to specify that explicitly:

```
/*
 * knexfile.mjs
 */
```

```
export default {
  migrations: {
    // ... client, connection,etc ....
    directory: './migrations',
    loadExtensions: ['.mjs'] //
  }
}
```

When using '.mjs' extensions for your knexfile and '.js' for the seeds/migrations, you will need to specify that explicitly.

```
/*
 * knexfile.mjs
 */
export default {
  migrations: {
    // ... client, connection,etc ....
    directory: './migrations',
    loadExtensions: ['.js'] // knex will search for 'mjs' file by default
  }
}
```

For the knexfile you can use a default export,
it will take precedence over named export.

```
/**
 * filename: knexfile.js
 * For the knexfile you can use a default export
 **/
export default {
  client: 'sqlite3',
  connection: {
    filename: '../test.sqlite3',
  },
  migrations: {
    directory: './migrations',
  },
  seeds: {
    directory: './seeds',
  },
}
```

```
/**
 * filename: knexfile.js
 * Let knex find the configuration by providing named exports,
 * but if exported a default, it will take precedence, and it will be used instead
 **/
const config = {
  client: 'sqlite3',
  connection: {
    filename: '../test.sqlite3',
  },
  migrations: {
    directory: './migrations',
  },
  seeds: {
    directory: './seeds',
  },
};
/** this will be used, it has precedence over named export */
export default config;
/** Named exports, will be used if you didn't provide a default export */
export const { client, connection, migrations, seeds } = config;
```

Seed and migration files need to follow Knex conventions

```
// file: seed.js
/**
 * Same as with the CommonJS modules
 * You will need to export a "seed" named function.
 * */
export function seed(next) {
  // ... seed logic here
}

// file: migration.js
/**
 * Same as the CommonJS version, the miration file should export
 * "up" and "down" named functions
 */
export function up(knex) {
  // ... migration logic here
}
export function down(knex) {
// ... migration logic here
}
```

Seed API

`knex.seed` is the class utilized by the knex seed CLI.

Each method takes an optional `config` object, which may specify the following properties:

- `directory`: a relative path to the directory containing the seed files. Can be an array of paths (default `./seeds`)
- `loadExtensions`: array of file extensions which knex will treat as seeds. For example, if you have typescript transpiled into javascript in the same folder, you want to execute only javascript seeds. In this case, set `loadExtensions` to `['.js']` (Notice the dot!) (default `['.co', '.coffee', '.eg', '.iced', '.js', '.litcoffee', '.ls', '.ts']`)
- `recursive`: if true, will find seed files recursively in the directory / directories specified
- `specific`: a specific seed file or an array of seed files to run from the seeds directory, if its value is `undefined` it will run all the seeds (default `undefined`). If an array is specified, seed files will be run in the same order as the array

- `sortDirsSeparately`: if true and multiple directories are specified, all seeds from a single directory will be executed before executing seeds from next folder (default `false`)
- `seedSource`: specify a custom seed source, see Custom Seed Source for more info (default filesystem)
- `extension`: extension to be used for newly generated seeds (default `js`)
- `timestampFilenamePrefix`: whether timestamp should be added as a prefix for newly generated seeds (default `false`)

Methods

make — `knex.seed.make(name, [config])`

Creates a new seed file, with the name of the seed file being added. If the seed directory config is an array of paths, the seed file will be generated in the latest specified.

run — `knex.seed.run([config])`

Runs all seed files for the current environment.

Custom seed sources

Knex supports custom seed sources, allowing you full control of where your seeds come from. This can be useful for custom folder structures, when bundling with webpack/browserify and other scenarios.

```
// Create a custom seed source class
class MySeedSource {
  // Must return a Promise containing a list of seeds.
  // Seeds can be whatever you want, they will be passed as
  // arguments to getSeed
  getSeeds() {
    // In this example we are just returning seed names
    return Promise.resolve(['seed1'])
  }

  getSeed(seed) {
    switch(seed) {
      case 'seed1':
        return (knex) => { /* ... */ }
    }
  }
}

// pass an instance of your seed source as knex config
knex.seed.run({ seedSource: new MySeedSource() })
```

Support

Have questions about the library? Come join us in the #bookshelf freenode IRC channel for support on knex.js and bookshelf.js, or post an issue on Stack Overflow or in the GitHub issue tracker.

F.A.Q.

How do I help contribute?
Glad you asked! Pull requests, or feature requests, though not always implemented, are a great way to help make Knex even better than it is now. If you're looking for something specific to help out with, there's a number of unit tests that aren't implemented yet, the library could never have too many of those. If you want to submit a fix or feature, take a look at the Contributing readme in the Github and go ahead and open a ticket.

How do I debug?
Knex is beginning to make use of the debug module internally, so you can set the `DEBUG` environment variable to `knex:*` to see all debugging, or select individual namespaces `DEBUG=knex:query,knex:tx` to constrain a bit.

If you pass `{debug: true}` as one of the options in your initialize settings, you can see all of the query calls being made. Sometimes you need to dive a bit further into the various calls and see what all is going on behind the scenes. I'd recommend node-inspector, which allows you to debug code with `debugger` statements like you would in the browser.

At the start of your application code will catch any errors not otherwise caught in the normal promise chain handlers, which is very helpful in debugging.

How do I run the test suite?
The test suite looks for an environment variable called `KNEX_TEST` for the path to the database configuration. If you run the following command:

```
$ export KNEX_TEST='/path/to/your/knex_config.js'
$ npm test
```

replacing with the path to your config file, and the config file is valid, the test suite should run properly.

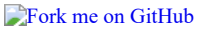
My tests are failing because slow DB connection and short test timeouts! How to extend test timeouts?
Sometimes, e.g. when running CI on travis, test suite's default timeout of 5 seconds might be too short. In such cases an alternative test timeout value in milliseconds can be specified using the `KNEX_TEST_TIMEOUT` environment variable.

```
$ export KNEX_TEST_TIMEOUT=30000
$ npm test
```

I found something broken with Amazon Redshift! Can you help?
Because there is no testing platform available for Amazon Redshift, be aware that it is included as a dialect but is unsupported. With that said, please file an issue if something is found to be broken that is not noted in the documentation, and we will do our best.

Change Log

1.0.3 — *11 February, 2022*
Bug fixes:



- Fix error message for missing migration files #4937
- Add withMaterialized and withNotMaterialized to method-constants #5009
- PostgreSQL: Fix whereJsonPath queries #5011
- PostgreSQL: Fix delete joins #5016
- CockroachDB: Fix whereJsonPath queries #5011
- MySQL: Create primary keys in same statement #5017

Typings:

- Fix type definition for getMigration in MigrationSource #4998
- Fix argument type of alter method #4996

Improvements:

- Use async / await syntax in seeds as default #5005

Documentation:

- Add Firebird dialect to ECOSYSTEM.md #5003

1.0.2 — *02 February, 2022*
New features:

- Support of MATERIALIZED and NOT MATERIALIZED with WITH/CTE #4940
- Add raw support in onConflict clause #4960
- Alter nullable constraint when alterNullable is set to true #4730
- Add alterType parameter for alter function #4967
- Support string json in json values #4988
- MySQL: add with clause #4508

Bug fixes:

- Fix error message for missing migration files #4937
- Move deferrable to after on update/on delete #4976
- Do not use sys.tables to find if a table exists #2328
- PostgreSQL: Fix Order nulls #4989
- MySQL: Fix collation when renaming column #2666
- SQLite: Same boolean handling in better-sqlite3 as in sqlite3 #4982

Typings:

- WhereILike - fix typo #4941

1.0.1 — *16 January, 2022*
Bug fixes:

- Fix package.json metadata

1.0.0 — *16 January, 2022*
Breaking changes

- Dropped support for Node 10;
- Replaced unsupported `sqlite3` driver with `@vscode/sqlite3`;
- Changed data structure from `RETURNING` operation to be consistent with `SELECT`;
- Changed Migrator to return list of migrations as objects consistently.

New features:

- Support fromRaw #4781
- Support zero precision in timestamp/datetime #4784
- Support whereLike and whereILike #4779
- Add JSDoc (TS flavor) to stub files #4809
- Allow skip binding in limit and offset #4811
- Support creating a new table in the database based on another table #4821
- Accept Raw on onIn joins #4830
- Implement support for custom seed sources #4842
- Add binary uuid option #4836
- ForUpdate array parameter #4882
- Add camel case to timestamps method #4803
- Advanced JSON support #4859
- Add type to TypeScript knexfile #4909
- Checks Constraints Support #4874
- Support creating multiple PKs with increments #4903
- Enable wrapIdentifier for SQLite .hasTable #4915
- MSSQL: Add support for unique constraint #4887
- SQLite: New dialect, using better-sqlite3 driver #4871
- SQLite: Switch to @vscode/sqlite3 #4866
- SQLite: Support createViewOrReplace #4856
- SQLite: Support RETURNING statements for better-sqlite3 driver #4934
- PostgreSQL: Support JOIN and USING syntax for Delete Statement #4800

Bug fixes:

- Fix overzealous warning on use of whereNot with "in" or "between" #4780
- Fix Union all + first syntax error #4799
- Make view columns optional in create view like #4829
- Insert lock row fix during migration #4865
- Fix for createViewOrReplace #4856
- SQLite: Fix foreign key constraints when altering a table #4189

- MySQL: Validate connection fix #4794
- MySQL: Set comment size warning limit to 1024 #4867

Typings:

- Allow string indexType in index creation #4791
- Add missing ints typings #4832
- Returning method types #4881
- Improve columnInfo type #4868

Show Full Changelog

0.95.15 — *22 December, 2021*
Bug fixes:

- Oracle:
- MariaDB: lock row fix during migration in MariaDB and Oracle #4865

0.95.14 — *09 November, 2021*
Bug fixes:

- MySQL: mysql2 dialect validate connection fix #4794

0.95.13 — *02 November, 2021*
Bug fixes:

- PostgreSQL: Support zero precision in timestamp/datetime #4784

Typings:

- Allow string indexType in index creation #4791

0.95.12 — *28 October, 2021*
New features:

- New dialect: CockroachDB #4742
- New dialect: pg-native #4327
- CockroachDB: add support for upsert #4767
- PostgreSQL: Support SELECT .. FOR NO KEY UPDATE / KEY SHARE row level locking clauses #4755
- PostgreSQL: Add support for 'CASCADE' in PostgreSQL 'DROP SCHEMA' queries #4713
- MySQL: Add storage engine index Type support to index() and unique() schema #4756
- MSSQL: Support table.primary, table.unique variant with options object #4710
- SQLite: Add setNullable support to SQLite #4684
- Add geometry column building #4776
- Add support for creating table copies #1373
- Implement support for views and materialized views #1626
- Implement partial index support #4768
- Support for 'is null' in 'order by' #3667

Bug fixes:

- Fix support for Oracle connections passed via knex.connection() #4757
- Avoid inserting multiple locks if a migration lock already exists #4694

Typings:

- Some TableBuilder methods return wrong types #4764
- Update JoinRaw bindings type to accept arrays #4752
- fix onDelete/onUpdate for ColumnBuilder #4656

0.95.11 — *03 September, 2021*
New features:

- Add support for nullability modification via schema builder (table.setNullable() and table.dropNullable()) #4657
- MySQL: Add support for mysql/mariadb-client JSON parameters in connectionURIs #4629
- MSSQL: Support comments as MS_Description properties #4632

Bug fixes:

- Fix Analytic orderBy and partitionBy to follow the SQL documentation #4602
- CLI: fix migrate:up for migrations disabling transactions #4550
- SQLite: Fix adding a column with a foreign key constraint in SQLite #4649
- MSSQL: columnInfo() support case-sensitive database collations #4633
- MSSQL: Generate valid SQL for withRecursive() #4514
- Oracle: withRecursive: omit invalid RECURSIVE keyword, include column list #4514

Improvements:

- Add .mjs migration and seed stubs #4631
- SQLite: Clean up DDL handling and move all operations to the parser-based approach #4648

0.95.10 — *20 August, 2021*
Improvements:

- Use sys info function instead of connection db name #4623

Typings:

- Deferrable and withkeyName should not be in ColumnBuilder #4600

0.95.9 — *31 July, 2021*
New features:

- Oracle: support specifying schema for dropTable and dropSequence #4596
- Oracle: support specifying schema for autoincrement #4594

Typings:

- Add TypeScript support for deferrable, new Primary/Unique syntax #4589

0.95.8 — *25 July, 2021*
New features:

- Add deferrable support for constraint #4584
- Implement delete with join #4568
- Add DPI error codes for Oracle #4536

Bug fixes:

- Fixing PostgreSQL datetime and timestamp column created with wrong format #4578

Typings:

- Improve analytic types #4576
- MSSQL: Add trustServerCertificate option #4500

0.95.7 — *10 July, 2021*
New features:

- Add ability to omit columns on an onConflict().ignore() #4557
- CLI: Log error message #4534

Typings:

- Export Knex.TransactionConfig #4498
- Include options object in count(Distinct) typings #4491
- Add types for analytic functions #4544

0.95.6 — *17 May, 2021*
Typings:

- Export TransactionProvider type #4489

0.95.5 — *11 May, 2021*
New features:

- SQLite: Add support for file open flags #4446
- Add .cjs extension to Seeder.js to support Node ESM #4381 #4382

Bug fixes:

- Remove peerDependencies to avoid auto-install on npm 7 #4480

Typings:

- Fix typing for increments and bigIncrements #4406
- Add typings for on JoinClause for onVal #4436
- Adding Type Definition for isTransaction #4418
- Export client class from knex namespace #4479

0.95.4 — *26 March, 2021*
Typings:

- Fix mistyping of stream #4400

0.95.3 — *25 March, 2021*
New features:

- PostgreSQL: Add "same" as operator #4372
- MSSQL: Improve an estimate of the max comment length #4362
- Throw an error if negative offset is provided #4361

Bug fixes:

- Fix timeout method #4324
- SQLite: prevent dropForeign from being silently ignored #4376

Typings:

- Allow config.client to be non-client instance #4367
- Add dropForeign arg type for single column #4363
- Update typings for TypePreservingAggregation and stream #4377

0.95.2 — *11 March, 2021*
New features:

- Improve ESM import support #4350

Bug fixes:

- CLI: update ts.stub files to new TypeScript namespace #4344
- CLI: fix TypeScript migration stub after 0.95.0 changes #4366

Typings:

- Move QueryBuilder and KnexTimeoutError into knex namespace #4358

Test / internal changes:

- Unify db test helpers #4356

0.95.1 — *04 March, 2021*
Bug fixes:

- CLI: fix `knex init` not finding default knexfile #4339

0.95.0 — *03 March, 2021*
New features:

- Add transaction isolation support #4185
- Add analytic functions #4188
- Change default to not trigger a promise rejection for transactions with a specified handler #4195
- Make toSQL().toNative() work for Raw to match the API for QueryBuilder #4058
- Allow 'match' operator #3569
- Support optimizer hints #4243
- Add parameter to prevent autoincrement columns from being primary keys #4266
- Make "first" and "pluck" mutually exclusive #4280
- Added merge strategy to allow selecting columns to upsert. #4252
- Throw error if the array passed to insert is empty #4289
- Events: introduce queryContext on query-error #4301
- CLI: Use UTC timestamp for new migrations #4245
- MSSQL: Replace MSSQL dialect with Tedious.js implementation #2857 #4281
- MSSQL: Use "nvarchar(max)" for ".json()" #4278
- MSSQL: Schema builder - add predictable constraint names for default values #4319
- MSSQL: Schema builder - attempt to drop default constraints when changing default value on columns #4321
- SQLite: Fallback to json for sqlite3 when using jsonb #4186
- SQLite: Return complete list of DDL commands for creating foreign keys #4194
- SQLite: Support dropping composite foreign keys #4202
- SQLite: Recreate indices when altering a table #4277
- SQLite: Add support for altering columns #4322

Bug fixes:

- Fix issue with .withSchema usage with joins on a subquery #4267
- Fix issue with schema usage with FROM clause contain QueryBuilder, function or Raw #4268
- CLI: Address raised security warnings by dropping liftoff #4122
- CLI: Fix an issue with npm@7 and ESM when `type` was set to `'module'` in `package.json` #4295
- PostgreSQL: Add check to only create native enum once #3658
- SQLite: Fix foreign key "on delete" when altering a table #4225
- SQLite: Made the constraint detection case-insensitive #4330
- MySQL: Keep auto increment after rename #4266
- MSSQL: don't raise query-error twice #4314
- MSSQL: Alter column must have its own query #4317

Typings:

- TypeScript 4.1+ is now required
- Add missing onConflict overrides #4182
- Introduce the "infamous triplet" export #4181
- Fix type definition of Transaction #4172
- Add typedefinitions for havingNotIn #4265
- Include 'name' property in MigratorConfig #4300
- Improve join and conflict types #4318
- Fix ArrayIfAlready type #4331

Test / internal changes:

- Drop global Knex.raw #4180
- Stop using legacy url.parse API #3702
- Various internal refactorings #4175 #4177 #4178 #4192
- Refactor to classes #4190 #4191 #4193 #4210 #4253
- Move transaction type tests to TSD #4208
- Clean up destroy logic #4248
- Colorize code snippets in readme files #4234
- Add "Ecosystem" documentation for Knex plugins #4183
- Documentation cleanup
- SQLite: Use SQLite "rename column" instead of a DDL helper #4200
- SQLite: Simplify reinsert logic when altering a table #4272

0.21.19 — *02 March, 2021*

SQLite: Made the constraint detection case-insensitive #4332

0.21.18 — *22 February, 2021*

CLI: Fix an issue with npm@7 and ESM when type was set to 'module' in package.json #4295

0.21.17 — *30 January, 2021*
Bug fixes:

- SQLite: Fix SQLite foreign on delete when altering a table #4261

New features:

- Add support for optimizer hints (see https://github.com/knex/documentation/pull/306 for documentation) #4243

0.21.16 — *17 January, 2021*
Bug fixes:

- MSSQL: Avoid passing unsupported pool param. Fixes node-mssql 7+ support #4236

0.21.15 — *26 December, 2020*
New features:

- SQLite: Add primary/foreign support on alterTable #4162
- SQLite: Add dropPrimary/dropForeign support on alterTable #4162

Typings:

- Add "after" and "first" to columnBuilder types #3549 #4169

Test / internal changes:

- Extract knex config resolution logic #4166
- Run CI using GitHub Actions #4168
- Add Node.js 15 to CI matrix #4173

0.21.14 — *18 December, 2020*
New features:

- MSSQL: support "returning" on inserts, updates and deletes on tables with triggers #4152
- Use esm import if package.json type is "module" #4158

Bug fixes:

- Make sure query-response and query-error events contain _knexTxId #4160

Test / internal changes:

- Improved integration test framework #4161

0.21.13 — *12 December, 2020*
New features:

- SQLite: Add support for `dropForeign` #4092
- Add support for WHERE clauses to "upsert" queries #4148

Bug fixes:

- MSSQL: Avoid connection getting stuck on socket hangup #4157
- Oracle: Support specifying non-default DB port #4147
- Oracle: Support inserts with only default values (empty body) #4092
- CLI: fix irregular seed file execution order #4156
- Fix performance of asyncStackTraces with enable-source-maps node flag #4154

Typings:

- PostgreSQL: Add support for application_name #4153
- Fix types for insert to allow array #4105
- Add types for userParams and withUserParams #4119
- Added type for withKeyName #4139
- Fix batchInsert definitions #4131
- Fix types for WhereIn signature (value or query builder) #3863
- Add types for connection config of mysql2 driver #4144

Test / internal changes:

- Move TS tests to tsd (WIP) #4109 #4110

0.21.12 — *02 November, 2020*
Typings:

- Reintroduce support for globally defining table/record mapping #4100
- Add a few missing types for MSSQL Connection #4103
- Make .ignore() and .merge() return QueryBuilder rather than QueryInterface #4102
- Use tarn config TS types instead of generic-pool #4064

0.21.11 — *01 November, 2020*
Typings:

- Revert support for globally defining table/record mapping #4099

0.21.10 — *31 October, 2020*
New features:

- Upsert support (Postgres/MySQL/Sqlite) #3763

Bug fixes:

- Switch to non-uuid knexQueryUids to avoid issues when mocking global date #4089

Typings:

- Allow to globally define table/record mapping #4071

0.21.9 — *27 October, 2020*
New features:

- add method clear(statement) to QueryBuilder #4051

Bug fixes:

- CLI: fix help text being printed twice #4072
- Oracle: columnInfo() no longer requires an Owner User #4053
- Add missing "start" event propagation from transaction #4087

0.21.8 — *27 October, 2020*
Bug fixes:

- MSSQL: Escape properly if literal '?' is needed #4053
- Make toQuery behavior consistent with pre-0.21.7 (do not break on empty builder) #4083
- Fix comment escaping for MySQL and PostgreSQL #4084

0.21.7 — *25 October, 2020*
New features:

- CLI: Add migration stub for .cjs extension #4065

Bug fixes:

- MSSQL: Add dynamic scaling for decimal values and prevents a UInt64 overflow #3910
- MSSQL: Fix apostrophe escaping #4077
- Ensure that semicolon is not appended to statements that already end with a semicolon #4052

Typings:

- Add arguments to QueryCallback in Where #4034

Test / internal changes:

- Replace lodash type-checks with native solutions #4056
- Replace mkdirp with native recursive flag #4060
- Replace inherits package with builtin utility #4059

0.21.6 — *27 September, 2020*
New features:

- CLI: New config parameter / CLI flag to prefixing seed filename with timestamp #3873
- CLI: throw an error when specific seed file cannot be found #4011
- Warn if whereNot is used with 'in' or 'between' #4038

Bug fixes:

- CLI: Fix double merging of config for migrator #4040

Typings:

- Unify SeedsConfig and SeederConfig #4003
- Allow string[] type for directory in SeedsConfig #4033

0.21.5 — *17 August, 2020*
New features:

- CLI: Improve Esm interop #3985
- CLI: Improve mjs module support #3980

Test / internal changes:

- Bump version of dtslint #3984
- Test/document esm interop mixed formats (knexfile/migrations/seeds) #3986

0.21.4 — *10 August, 2020*
New features:

- CLI: Add new option for seed: recursive #3974

Bug fixes:

- CLI: Do not load seeds from subfolders recursively by default #3974

0.21.3 — *08 August, 2020*
New features:

- CLI: Support multiple directories for seeds #3967

Bug fixes:

- Ensure DB stream is destroyed when the PassThrough is destroyed #2324
- Support postProcessResponse for streams #3931
- Fix ESM module interop for calling module/package of type 'module' #3938
- CLI: Fix migration source name in rollback all #3956

- Fix getMergedConfig calls to include client logger #3920
- Escape single quoted values passed to defaultTo function #3899

Typings:

- Add .timeout(ms) to .raw()'s typescript typings #3885
- Add typing for double table column builder #3950
- Add a phantom tag to Ref type to mark received type parameters as used #3934
- Add `null` as valid binding type #3946

Test / internal changes:

- Change query lab link to https #3933

0.21.2 — *10 July, 2020*
New features:

- Warn user if custom migration source is being reset #3839
- Prefer `void` as return type on migration generator ts stub #3865
- MSSQL: Added the removal of a columns default constraint, before dropping the column #3855

Typings:

- Fix definition for raw querybuilders #3846

Test / internal changes:

- Refactor migration logic to use async/await #3838

0.21.1 — *28 April, 2020*
New features:

- CLI: Add migrate:unlock command, truncate on forceFreeMigrationsLock #3822
- CLI: Add support for cjs files by default #3829

Bug fixes:

- CLI: Fix inference of seed/migration extension from knexfile extension #3814
- rewrite delay to not node-only version. Fixes compatibility with browsers #3820

Test / internal changes:

- Update dependencies. Explicitly support Node.js 14 #3825 #3830

0.21.0 — *18 April, 2020*
Improvements

- Reduce size of lodash in bundle #3804

Breaking changes

- Dropped support for Node 8
- Breaking upstream change in `pg-query-stream`: `Changed stream.close to stream.destroy which is the official way to terminate a readable stream. This is a breaking change if you rely on the stream.close method on pg-query-stream...though should be just a find/replace type operation to upgrade as the semantics remain very similar (not exactly the same, since internals are rewritten, but more in line with how streams are "supposed" to behave).`

Test / internal changes:

- Updated Tarn.js to a version 3.0.0
- Updated mkdirp to a version 1.0.4
- Updated examples to use ES2015 style #3810

0.20.15 — *16 April, 2020*
Bug fixes:

- Support for `.finally(..)` on knex's Promise-alikes #3800

Typings:

- Add types for `.distinctOn` #3784

0.20.14 — *13 April, 2020*
New features:

- CLI: adds support for asynchronous knexfile loading #3748
- Add clearGroup method #3771

Typings:

- Support Raw types for insert, where, update #3730
- Add typings for MigrationSource #3756
- Update signature of orderBy to support QueryBuilder inside array #3757
- Add toSQL and toString to SchemaBuilder #3758
- `interface Knex` and `function Knex` should have the same types #3787
- Fix minor issues around typings #3765

Test / internal changes:

- Minor test internal enhancements #3747

- Minor improvements on the usage of fs utilities #3749
- Split tests in groups #3785

0.20.13 — *23 March, 2020*
Bug fixes:

- Correctly handle dateToString escaping without timezone passed #3742
- Make protocol length check more defensive #3744

Typings:

- Make the ChainableInterface conform to Promise #3724

0.20.12 — *19 March, 2020*
Bug fixes:

- Added missing call to _reject in Transactor#transaction #3706
- Fix method binding on knex proxy #3717
- Oracle: Transaction_OracleDB can use config.connection #3731

Typings:

- Fix incorrect type signature of Having #3719

Test / internal changes:

- Cleanup/remove transaction stalling #3716
- Rewrote Transaction#acquireConnection() methods to use async #3707

0.20.11 — *26 February, 2020*
Breaking changes:

- Knex returns native JS promises instead of Bluebird ones. This means that you no longer use such methods as `map`, `spread` and `reduce` on QueryBuilder instance.

New features:

- Oracle: Add OracleDB handling for buffer type in fetchAsString #3685

Bug fixes:

- Fix race condition in non-container transactions #3671

Typings:

- Mark knex arguments of composite/collection types to be readonly #3680

Test / internal changes:

- Remove dependency on Bluebird methods from sources #3683
- Cleanup and extract Transaction Workflow logic #3674

0.20.10 — *13 February, 2020*
Bug fixes:

- Oracle: commit was a no-op causing race conditions #3668
- CLI: Knex calls process.chdir() before opening Knexfile #3661
- Fixed unresolved promise in cancelQuery() #3666

Typings:

- `fn.now` takes optionally a precision argument. #3662
- PG: Include SSL in connection definition #3659

Test / internal changes:

- replace Bluebird.timeout #3634

0.20.9 — *08 February, 2020*
Bug fixes:

- CLI: Improve Support for Liftoff's Preloaders - this should fix some cases like using TS for your migrations #3613

Typings:

- MSSQL: Add `enableArithAbort` to `MsSqlConnectionConfig`

Test / internal changes:

- Refactor more tests to use cli-testlab #3640
- Update QueryCompiler implementation to use classes #3647

0.20.8 — *14 January, 2020*
New features:

- CLI: Support ES6 modules via flag --esm #3616

Bug fixes:

- CLI: Print help only when there are no arguments #3617

Typings:

- Fix incorrect type of QueryBuilder.first('*') result #3621

0.20.7 — *07 January, 2020*
New features:

- Throw better error when trying to modify schema while using unsupported dialect #3609

Bug fixes:

- Oracle: dispose connection on connection error #3611
- Oracle: fix not releasing connection from pool on disconnect #3605
- CLI: prevent warning with root command #3604

Typings:

- Add create/drop schema methods to SchemaBuilder #3579

0.20.6 — *29 December, 2019*
Bug fixes:

- Enforce Unix (lf) line terminators #3598

0.20.5 — *29 December, 2019*
New features:

- Return more information about empty updates #3597

Bug fixes:

- Fix colors in debug logs #3592

Test / internal changes:

- Use more efficient algorithm for generating internal ids #3595 #3596
- Use Buffer.alloc() instead of deprecated constructor #3574

0.20.4 — *08 December, 2019*
Bug fixes:

- Fix debug logger messing up queries with % #3566
- Make logger methods mutually consistent #3567

Typings:

- Add missing methods to client type #3565
- Fix queryContext function defintion #3562
- Fix QueryBuilder.extend this type #3526 #3528

Test / internal changes:

- Remove bluebird.using #3552

0.20.3 — *27 November, 2019*
New features:

- MSSQL, MySQL: Add connection string qs to connection params #3547

Bug fixes:

- Oracle: Fix issue retrieving BLOB from database #3545
- PostgreSQL: Timeout for postgresql use cancel instead of terminate #3518
- Make sure CLI works for namespaced knex packages #2539

Typings:

- Lift up dialect specific methods in the CreateTableBuilder #3532
- Add client property to QueryBuilder type #3541
- Support 'only' option #3551

0.20.2 — *14 November, 2019*
New features:

- Add support for distinct on for postgres #3513

Bug fixes:

- Make sqlite3 hasColumn case insensitive #3435

Typings:

- Fix PoolConfig typing #3505
- Expand SeedsConfig types #3531
- Make the default type parameters of QueryBuilder less strict #3520
- Fix regression in older version of node when Promise#finally was not available #3507

0.20.1 — *29 October, 2019*
New features:

- Declare drivers as optional peerDependencies #3081

- Dynamic connection configuration resolution #3497

Bug fixes:

- Wrap subQuery with parenthesis when it appears as table name #3496
- Fix Oracle error codes #3498

Typings:

- Add interface for PG Connection object #3372
- Gracefully handle global promise pollution #3502

0.20.0 — *25 October, 2019*
New features:

- orderBy accepts QueryBuilder #3491
- Add validation in `.offset()` #2908
- disable_migrations_list_validation feature #3448

Bug fixes:

- Fix oracledb driver v4 support #3480
- Fix some issues around seed and migration generation #3479
- Fix bugs in replacement logic used when dropping columns in SQLite #3476

Typings:

- Add types to the Migrator interface #3459
- Fix typings of index and dropIndex TableBuilder methods #3486
- Fixes types for Seeder#run #3438

Test / internal changes:

- Execute CI on Node.js 13
- Bluebird: remove usage of `return`, `reflect`, `fromCallback` methods #3483
- Bluebird: remove Bluebird.bind #3477
- Bluebird: use util.promisify instead of Bluebird.promisify #3470
- Bluebird: remove Bluebird.each #3471
- Bluebird: remove Bluebird.map and Bluebird.mapSeries #3474
- Bluebird: replace Bluebird.map with Promise.all #3469
- Update badges #3482

0.19.5 — *06 October, 2019*
New features:

- CLI: Migrations up/down commands - filename parameter #3416
- Oracle: Support stored procedures #3449

Bug fixes:

- MSSQL: Escape column ids correctly in all cases (reported by Snyk Security Research Team) #3382
- SQLite: Fix handling of multiline SQL in SQLite3 schema #3411
- Fix concurrent child transactions failing #2213 #3440

Typings:

- Add missing Migrator.list typing #3460
- Fix Typescript type inference for to better support wildcard (*) calls #3444
- Make options argument optional in timeout #3442

Test / internal changes:

- Enable linting in CI #3450

0.19.4 — *09 September, 2019*
New features:

- Add undefined columns to undefined binding(s) error #3425

Typings:

- Add `specific` to SeederConfig type #3429
- Fix some issues with QueryBuilder types #3427

0.19.3 — *25 August, 2019*
Bug fixes:

- Fix migrations for native enums to use table schema #3307

New features:

- Add ability to manually define schema for native enums #3307
- Add SSL/TLS support for Postgres connection string #3410
- CLI: new command that lists all migrations with status #3390

Typings:

- Include schemaName in EnumOptions #3415
- Allow `ColumnBuilder.defaultTo()` to be `null` #3407

Changes:

- migrate: Refactor _lockMigrations to avoid forUpdate - makes migrations compatible with CockroachDB #3395

0.19.2 — *17 August, 2019*
Changes:

- Make transaction rejection consistent across dialects #3399
- More consistent handling of nested transactions #3393

New features:

- Fallback to JSON when using JSONB in MySQL #3394

0.19.1 — *23 July, 2019*
New features:

- Allow to extend knex query builder #3334
- Add .isCompleted() to transaction #3368
- Minor enhancements around aliasing of aggregates #3354

Typings:

- Update configuration typings to allow for oracle db connectionstring #3361
- Update Knex.raw type to be any by default because the actual type is dialect specific #3349

0.19.0 — *11 July, 2019*
Changes:

- Pooling: tarn.js connection pool was updated to version 2.0.0. This fixes issue with destroying connections and introduces support for connection pool event handlers. Please see tarn.js documentation for more details #3345
- Pooling: Passing unsupported pooling configuration options now throws an error
- Pooling: `beforeDestroy` configuration option was removed

0.18.4 — *10 July, 2019*
New features:

- Seeds: Option to run specific seed file #3335
- Implement "skipLocked()" and "noWait()" #2961

Bug fixes:

- CLI: Respect the knexfile stub option while generating a migration #3337
- Fix mssql import not being ignored, breaking webpack builds #3336

0.18.3 — *04 July, 2019*
New features:

- CLI: add --stub option to migration:make #3316

Bug fixes:

- Fix return duplicate transaction promise for standalone transactions #3328

0.18.2 — *03 July, 2019*
Bug fixes:

- Fix remove duplicate transaction rejection #3324
- Fix issues around specifying default values for columns #3318
- CLI: Fix empty --version output #3312

0.18.1 — *30 June, 2019*
Bug fixes:

- Do not reject duplicate promise on transaction rollback #3319

0.18.0 — *26 June, 2019*
Bug fixes:

- Do not reject promise on transaction rollback (by default only for new, non-callback, style of transactions for now to avoid breaking old code) #3235

New features:

- Added `doNotRejectOnRollback` options for starting transactions, to prevent rejecting promises on rollback for callback-style transactions.
- Use extension from knexfile for generating migrations unless overriden #3282
- Use migrations.extension from config when generating migration #3242
- Expose executionPromise for transactors #3297

Bug fixes:

- Oracle: Updated handling of connection errors for disposal #2608
- Fix extension resolution from env configs #3294

Test / internal changes:

- Drop support for Node.js 6 #3227
- Remove Babel #3227
- Remove Bluebird #3290 #3287 #3285 #3267 #3266 #3263
- Fix comments that were modified by find & replace #3308

Typings:

- Add workarounds for degraded inference when strictNullChecks is set to false #3275
- Add stub type definition for Migrator config #3279
- Add stub to seeds type #3296
- Fix MSSQL config typings #3269
- Add pgsql specific table builder method typings #3146

0.17.5 — *8 June, 2019*
Typings:

- Include result.d.ts in published package #3271

0.17.4 — *8 June, 2019*
Typings:

- Fix some cases of left-to-right inference causing type mismatch #3265
- Improve count typings #3249

Bug fixes:

- Fix error message bubbling up on seed error #3248

0.17.3 — *2 June, 2019*
Typings:

- Improve typings for aggregations #3245
- Add decimalNumbers to MySqlConnectionConfig interface #3244

0.17.2 — *1 June, 2019*
Typings

- Improve count typings #3239

Bug fixes:

- "colorette" dependency breaks browserify builds #3238

0.17.1 — *31 May, 2019*
New features:

- Add migrate:down functionality #3228

Typings:

- Update type of aggregation results to not be arrays when first has been invoked before #3237
- Include undefined in type of single row results #3231
- Fix incorrect type definitions for single row queries #3230

0.17.0 — *28 May, 2019*
New features:

- Add support for returning started transaction without immediately executing it #3099
- Add support for passing transaction around with only starting it when needed #3099
- Add clearHaving function #3141
- Add --all flag for rollback in CLI #3187
- Add error detail log to knex CLI #3149
- Support multi-column whereIn in sqlite through values clause #3220
- Allow users to specify the migrations "tableName" parameter via the CLI #3214
- Unify object options handling for datetime/timestamp across dialects #3181
- Add "up" command for migrations #3205

Typings:

- Add default values for generic types (fixes backwards compatibility broken by 0.16.6) #3189
- Make function types generic in type definitions #3168
- Add missing types to MigratorConfig #3174
- Add types for havingBetween, orHavingBetween, havingNotBetween and orHavingNotBetween #3144
- Update Knex.Config types to include log #3221
- Fix some more cases of missing typings #3223
- Support type safe refs #3215
- Expose some utility types #3211
- Fix issues with typings of joins and some conflicts with Bluebird typings #3209

Bug fixes:

- Fix order of migration rollback #3172

Test / internal changes:

- Execute CI tests on Node.js 12 #3171
- Docker-based test dbs #3157
- Use cli-testlab for testing CLI #3191

0.16.5 — *11 Apr, 2019*

Bundle polyfills with knex for 0.16.x line again #3139

0.16.4 — *11 Apr, 2019*
New features:

- Boolean param for rollback() to rollback all migrations #2968
- seed:run print the file name of the failing seed #2972 #2973
- verbose option to CLI commands #2887
- add intersect() #3023
- Improved format for TS stubs #3080
- MySQL: Support nullable timestamps #3100
- MySQL: Warn `.returning()` does not have any effect #3039

Bug fixes:

- Respect "loadExtensions" configuration #2969
- Fix event listener duplication when using Migrator #2982
- Fix fs-migrations breaking docs #3022
- Fix sqlite3 drop/renameColumn() breaks with postProcessResponse #3040
- Fix transaction support for migrations #3084
- Fix queryContext not being passed to raw queries #3111
- Typings: Allow to pass query builders, identifiers and raw in various places as parameters #2960
- Typings: toNative() definition #2996
- Typings: asCallback() definition #2963
- Typings: queryContext() type definition Knex.Raw #3002
- Typings: Add "constraintName" arg to primary() definition #3006
- Typings: Add missing schemaName in MigratorConfig #3016
- Typings: Add missing supported parameter types and toSQL method #2960
- Typings: Update enum arguments to reflect latest signature #3043
- Typings: Add size parameter to integer method #3074
- Typings: Add 'string' as accepted Knex constructor type definition #3105
- Typings: Add boolean as a column name in join #3121
- Typings: Add missing clearOrder & clearCounters types #3109
- Dependencies: Fix security warning #3082
- Do not use unsupported column width/length arguments on data types int and tinyint in MSSQL #2738

Changes:

- Make unionAll()'s call signature match union() #3055

Test / internal changes:

- Swap chalk→colorette / minimist→getopts #2718
- Always use well documented pg client query() config argument #3004
- Do not bundle polyfills with knex #3024

0.16.3 — *19 Dec, 2018*
Bug fixes:

- @babel/polyfill loaded multiple times #2955
- Resolve migrations and seeds relatively to knexfile directory when specified (the way it used to be before 0.16.1) #2952

0.16.2 — *10 Dec, 2018*
Bug fixes:

- Add TypeScript types to the "files" entry so they are properly included in the release #2943

0.16.1 — *28 Nov, 2018*
Breaking Changes:

- Use datetime2 for MSSQL datetime + timestamp types. This change is incompatible with MSSQL older than 2008 #2757
- Knex.VERSION() method was removed, run "require('knex/package').version" instead #2776
- Knex transpilation now targets Node.js 6, meaning it will no longer run on older Node.js versions #2813
- Add json type support for SQLite 3.9+ (tested to work with Node package 'sqlite3' 4.0.2+) #2814

New features:

- Support passing explicit connection to query builder (#2817)
- Introduced abstraction for getting migrations to make migration bundling easier #2775
- Allow timestamp with timezone on mssql databases #2724
- Allow specifying multiple migration directories #2735
- Allow cloning query builder with .userParams({}) assigned to it #2802
- Allow chaining of increment, decrement, and update #2740
- Allow table names with `forUpdate`/`forShare` #2834
- Added `whereColumn` and the associated `not` / `and` / `or` methods for using columns on the right side of a where clause #2837
- Added `whereRecursive` method to make self-referential CTEs possible #2889
- Added support for named unique, primary and foreign keys to SQLite3 #2840
- Added support for generating new migration and seed files without knexfile #2884 #2905 #2935
- Added support for multiple columns in `.orderBy()` #2881
- Added option of `existingType` to `.enum()` method to support repeated use of enums #2719
- Added option to pass `indexType` for MySQL dialect #2890
- Added `onVal` and the associated `not` / `and` / `or` methods for using values in `on` clauses within joins #2746
- Kill queries after timeout for PostgreSQL #2636
- Manage TypeScript types internally #2845
- Support 5.0.0+ versions of mssql driver #2861
- Typescript migration stub #2816
- Options object for passing timestamp parameters + regression tests #2919

Bug fixes:

- Implement fail-fast logic for dialect resolution #2776
- Fixed identifier wrapping for `using()`. Use columnize instead of wrap in using() #2713
- Fix issues with warnPromise when migration does not return a promise #2730

- Compile with before update so that bindings are put in correct order #2733
- Fix join using builder withSchema #2744
- Throw instead of process.exit when client module missing #2843
- Display correct filename of a migration that failed #2910
- Fixed support of knexSnakeCaseWrappers in migrations #2914
- SQlite3 renameColunm quote fix #2833
- Adjust typing for forUpdate()/forShare() variant with table names #2858
- Fix execution of Oracle tests on Node 11 #2920
- Fix failures in oracle test bench and added it back to mandatory CI tests #2924
- Knex client knexfile resolution fix #2923
- Add queryContext to type declarations #2931

Test / internal changes:

- Add tests for multiple union arguments with callbacks and builders #2749
- Update dependencies #2772 #2810 #2842 #2848 #2893 #2904
- Separate migration generator #2786
- Do not postprocess internal queries in Migrator #2914 #2934
- Use Babel 7 #2813
- Introduce LGTM.com badge #2755
- Cleanup based on analysis by https://lgtm.com #2870
- Add test for retrieving null dates #2865
- Add link to wiki #2866
- Add tests for specifying explicit pg version #2895
- Execute tests on Node.js 11 #2873
- Version upgrade guide #2894

0.16.0 — *27 Nov, 2018*
Changes:

- THIS RELEASE WAS UNPUBLISHED FROM NPM BECAUSE IT HAD BROKEN MIGRATIONS USING `postprocessResponse` FEATURE (#2644)

0.15.2 — *19 Jul, 2018*
Changes:

- Rolled back changes introduced by #2542, in favor of opt-in behavior by adding a precision option in `date` / `timestamp` / `datetime` / `knex.fn.now` (#2715, #2721)

0.15.1 — *12 Jul, 2018*
Bug fixes:

- Fix warning erroneously displayed for mysql #2705

0.15.0 — *1 Jul, 2018*
Breaking Changes:

- Stop executing tests on Node 4 and 5. #2451 (not supported anymore)
- `json` data type is no longer converted to `text` within a schema builder migration for MySQL databases (note that JSON data type is only supported for MySQL 5.7.8+) #2635
- Removed WebSQL dialect #2461
- Drop mariadb support #2681
- Primary Key for Migration Lock Table #2569. This shouldn't affect to old loc tables, but if you like to have your locktable to have primary key, delete the old table and it will be recreated when migrations are ran next time.
- Ensure knex.destroy() returns a bluebird promise #2589
- Increment floats #2614
- Testing removal of 'skim' #2520, Now rows are not converted to plain js objects, returned row objects might have changed type with oracle, mssql, mysql and sqlite3
- Drop support for strong-oracle #2487
- Timeout errors doesn't silently ignore the passed errors anymore #2626
- Removed WebSQL dialect #2647
- Various fixes to mssql dialect to make it compatible with other dialects #2653, Unique constraint now allow multiple null values, float type is now float instaed of decimal, rolling back transaction with undefined rejects with Error, select for update and select for share actually locks selected row, so basically old schema migrations will work a lot different and produce different schema like before. Also now MSSQL is included in CI tests.

Bug fixes:

- Fixes onIn with empty values array #2513
- fix wrapIdentifier not being called in postgres alter column #2612
- fixes wrapIdentifier to work with postgres `returning` statement 2630 #2642
- Fix mssql driver crashing in certain cases when conneciton is closed unexpectedly #2637
- Removed semicolon from rollback stmt for oracle #2564
- Make the stream catch errors in the query #2638

New Features:

- Create timestamp columns with microsecond precision on MySQL 5.6 and newer #2542
- Allow storing stacktrace, where builder is initialized to be able trace back where certain query was created #2500 #2505
- Added 'ref' function #2509, no need for knex.raw('??', ['id']) anymore, one can do knex.ref('id')
- Support postgresql connection uri protocol #2609
- Add support for native enums on Postgres #2632
- Allow overwriting log functions #2625

Test / internal changes:

- chore: cache node_modules #2595
- Remove babel-plugin-lodash #2634
- Remove readable-stream and safe-buffer #2640
- chore: add Node.js 10 #2594

- add homepage field to package.json #2650

0.14.6 — *12 Apr, 2018*
Bug fixes:

- Restored functionality of query event #2566 (#2549)

0.14.5 — *8 Apr, 2018*
Bug fixes:

- Fix wrapping returning column on oracledb #2554

New Features:

- Support passing DB schema name for migrations #2499 #2559
- add clearOrder method #2360 #2553
- Added knexTxId to query events and debug calls #2476
- Support multi-column `whereIn` with query #1390
- Added error if chaining update/insert/etc with first() #2506
- Checks for an empty, undefined or null object on transacting #2494
- countDistinct with multiple columns #2449

Test / internal changes:

- Added npm run test:oracledb command that runs oracledb tests in docker #2491
- Runnin mssql tests in docker #2496
- Update dependencies #2561

0.14.4 — *19 Feb, 2018*
Bug fixes:

- containsUndefined only validate plain objects. Fixes #1898 (#2468)
- Add warning when using .returning() in sqlite3. Fixes #1660 (#2471)
- Throw an error if .update() results in an empty sql (#2472)
- Removed unnecessary createTableIfNotExist and replaced with createTable (#2473)

New Features:

- Allow calling lock procedures (such as forUpdate) outside of transaction. Fixes #2403. (#2475)
- Added test and documentation for Event 'start' (#2488)

Test / internal changes:

- Added stress test, which uses TCP proxy to simulate flaky connection #2460
- Removed old docker tests, new stress test setup (#2474)
- Removed unused property __cid on the base client (#2481)
- Changed rm to rimraf in 'npm run dev' (#2483)
- Changed babel preset and use latest node as target when running dev (#2484)

0.14.3 — *8 Feb, 2018*
Bug fixes:

- Use tarn as pool instead of generic-pool which has been given various problems #2450
- Fixed mysql issue where add columns failed if using both after and collate #2432
- CLI sets exit-code 1 if the command supplied was not parseable #2358
- Set toNative() to be not enumerable #2388
- Use wrapIdentifier in columnInfo. fixes #2402 #2405
- Fixed a bug when using .returning (OUTPUT) in an update query with joins in MSSQL #2399
- Better error message when running migrations fail before even starting run migrations #2373
- Read oracle's UV_THREADPOOL_SIZE env variable correctly #2372
- Added decimal variable precision / scale support #2353

New Features:

- Added queryContext to schema and query builders #2314
- Added redshift dialect #2233
- Added warning when one uses .createTableIfNotExist and deprecated it from docs #2458

Test / internal changes:

- Update dependencies and fix ESLint warnings accordingly #2433
- Disable oracledb tests from non LTS nodes #2407
- Update dependencies #2422

0.14.2 — *24 Nov, 2017*
Bug fixes:

- Fix sqlite3 truncate method to work again #2348

0.14.1 — *19 Nov, 2017*
Bug fixes:

- Fix support for multiple schema names in in postgres `searchPath` #2340
- Fix create new connection to pass errors to query instead of retry loop #2336
- Fix recognition of connections closed by server #2341

0.14.0 — *6 Nov, 2017*
Breaking Changes:

- Remove sorting of statements from update queries #2171

- Updated allowed operator list with some missing operators and make all to lower case #2239
- Use node-mssql 4.0.0 #2029
- Support for enum columns to SQlite3 dialect #2055
- Better identifier quoting in Sqlite3 #2087
- Migration Errors - Display filename of of failed migration #2272

Other Features:

- Post processing hook for query result #2261
- Build native SQL where binding parameters are dialect specific #2237
- Configuration option to allow override identifier wrapping #2217
- Implemented select syntax: select({ alias: 'column' }) #2227
- Allows to filter seeds and migrations by extensions #2168
- Reconnecting after database server disconnect/reconnect + tests #2017
- Removed filering from allowed configuration settings of mysql2 #2040
- Allow raw expressions in query builder aggregate methods #2257
- Throw error on non-string table comment #2126
- Support for mysql stream query options #2301

Bug fixes:

- Allow update queries and passing query builder to with statements #2298
- Fix escape table name in SQLite columnInfo call #2281
- Preventing containsUndefined from going to recursion loop #1711
- Fix error caused by call to knex.migrate.currentVersion #2123
- Upgraded generic-pool to 3.1.7 (did resolve some memory issues) #2208
- Allow using NOT ILIKE operator #2195
- Fix postgres searchPath to be case-sensitive #2172
- Fix drop of multiple columns in sqlite3 #2107
- Fix adding multiple columns in Oracle #2115
- Use selected schema when dropping indices in Postgres. #2105
- Fix hasTable for MySQL to not do partial matches #2097
- Fix setting autoTransaction in batchInsert #2113
- Fix connection error propagation when streaming #2199
- Fix comments not being applied to increments columns #2243
- Fix mssql wrong binding order of queries that combine a limit with select raw or update #2066
- Fixed mysql alter table attributes order #2062

Test / internal changes:

- Update each out-of-date dependency according to david-dm.org #2297
- Update v8flags to version 3.0.0 #2288
- Update interpret version #2283
- Fix debug output typo #2187
- Docker CI tests #2164
- Unit test for right/rightOuterJoin combination #2117
- Unit test for fullOuterJoin #2118
- Unit tests for table comment #2098
- Test referencing non-existent column with sqlite3 #2104
- Unit test for renaming column in postgresql #2099
- Unit test for cross-join #2102
- Fix incorrect parameter name #2068

0.13.0 — *29 Apr, 2017*
Breaking Changes:

- Multiple concurrent migration runners blocks instead of throwing error when possible #1962
- Fixed transaction promise mutation issue #1991

Other Changes:

- Allow passing version of connected db in configuration file #1993
- Bugfixes on batchInsert and transactions for mysql/maria #1992
- Add fetchAsString optional parameter to oracledb dialect #1998
- fix: escapeObject parameter order for Postgres dialect. #2003

0.12.9 — *23 Mar, 2017*

Fixed unhandled exception in batchInsert when the rows to be inserted resulted in duplicate key violation #1880

0.12.8 — *15 Mar, 2017*

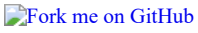
- Added clearSelect and clearWhere to query builder #1912
- Properly close Postgres query streams on error #1935
- Transactions should never reject with undefined #1970
- Clear acquireConnectionTimeout if an error occurs when acquiring a connection #1973

0.12.7 — *17 Feb, 2017*
Accidental Breaking Change:

- Ensure that 'client' is provided in knex config object #1822

Other Changes:

- Support custom foreign key names #1311, #1726
- Fixed named bindings to work with queries containing :-chars #1890
- Exposed more promise functions #1896
- Pass rollback errors to transaction promise in mssql #1885
- ONLY keyword support for PostgreSQL (for table inheritance) #1874



- Fixed Mssql update with join syntax #1777
- Replace migrations and seed for react-native packager #1813
- Support knexfile, migration and seeds in TypeScript #1769
- Fix float to integer conversion of decimal fields in MSSQL #1781
- External authentication capability when using oracledb driver #1716
- Fixed MSSQL incorect query build when locks are used #1707
- Allow to use `first` method as aliased select #1784
- Alter column for nullability, type and default value #46, #1759
- Add more having* methods / join clause on* methods #1674
- Compatibility fixes and cleanups #1788, #1792, #1794, #1814, #1857, #1649

0.12.6 — *19 Oct, 2016*

- Address warnings mentioned in #1388 (#1740)
- Remove postinstall script (#1746)

0.12.5 — *12 Oct, 2016*

- Fix broken 0.12.4 build (removed from npm)
- Fix #1733, #920, incorrect postgres array bindings

0.12.3 — *9 Oct, 2016*

- Fix #1703, #1694 - connections should be returned to pool if acquireConnectionTimeout is triggered
- Fix #1710 regression in postgres array escaping

0.12.2 — *27 Sep, 2016*

- Restore pool min: 1 for sqlite3, #1701
- Fix for connection error after it's closed / released, #1691
- Fix oracle prefetchRowCount setting, #1675

0.12.1 — *16 Sep, 2016*

- Fix MSSQL sql execution error, #1669
- Added DEBUG=knex:bindings for debugging query bindings, #1557

0.12.0 — *13 Sep, 2016*

- Remove build / built files, #1616
- Upgrade to Babel 6, #1617
- Reference Bluebird module directly, remove deprecated .exec method, #1618
- Remove documentation files from main repo
- Fix broken behavior on WebSQL build, #1638
- Oracle id sequence now handles manual inserts, #906
- Cleanup PG escaping, fix #1602, #1548
- Added `with` to builder for common table expressions, #1599
- Fix #1619, pluck with explicit column names
- Switching back to generic-pool for pooling resource management
- Removed index.html, please direct all PR's for docs against the files in knex/documentation

0.11.10 — *9 Aug, 2016*

- Added CHANGELOG.md for a new documentation builder coming soon, #1615
- Minor documentation tweaks
- PG: Fix Uint8Array being considered undefined, #1601
- MSSQL: Make columnInfo schema dynamic, #1585

0.11.9 — *21 Jul, 2016*

Reverted knex client breaking change (commit b74cd69e906), fixes #1587

0.11.8 — *21 Jul, 2016*

- Oracledb dialect #990
- Documentation fix #1532
- Allow named bindings to be escaped. #1576
- Several bugs with MS SQL schema creation and installing from gihub fix #1577
- Fix incorrect escaping of backslashes in SqlString.escape #1545

0.11.7 — *19 Jun, 2016*

Add missing dependency. #1516

0.11.6 — *18 Jun, 2016*

- Allow cancellation on timeout (MySQL) #1454
- Better bigint support. (MSSQL) #1445
- More consistent handling of `undefined` values in `QueryBuilder#where` and `Raw`. #1459
- Fix Webpack build. #1447
- Fix code that triggered Bluebird warnings. #1460, #1489
- Fix `ping` function. (Oracle) #1486
- Fix `columnInfo`. (MSSQL) #1464
- Fix `ColumnCompiler#binary`. (MSSQL) #1464
- Allow connection strings that do not contain a password. #1473
- Fix race condition in seed stubs. #1493
- Give each query a UUID. #1510

0.11.5 — *26 May, 2016*

Bugfix: Using `Raw` or `QueryBuilder` as a binding to `Raw` now works as intended

0.11.4 — *22 May, 2016*

- Bugfix: Inconsistency of `.primary()` and `.dropPrimary()` between dialects #1430
- Feature: Allow using custom Client/Dialect (you can pass your own client in knex config) #1428
- Docs: Add documentation for .dropTimestamps #1432
- Bugfix: Fixed passing undefined fields for insert/update inside transaction #1423
- Feature: `batchInsert` with existing transaction #1354
- Build: eslint instead of jshint #1416
- Bugfix: Pooled connections not releasing #1382
- Bugfix: Support passing `knex.raw` to `.whereNot` #1402
- Docs: Fixed list of dialects which supports `.returning` #1398
- Bugfix: rename table does not fail anymore even with schema defined #1403

0.11.3 — *14 May, 2016*

Support nested joins. #1397

0.11.2 — *14 May, 2016*

- Prevent crash on `knex seed:make`. #1389
- Improvements to `batchInsert`. #1391
- Improvements to inserting `DEFAULT` with `undefined` binding. #1396
- Correct generated code for adding/dropping multiple columns. (MSSQL) #1401

0.11.1 — *6 May, 2016*

Fix error in CLI command `migrate:make`. #1386

0.11.0 — *5 May, 2016*
Breaking Changes:

- `QueryBuilder#orWhere` joins multiple arguments with `AND`. #1164

Other Changes:

- Collate for columns. (MySQL) #1147
- Add `QueryBuilder#timeout`, `Raw#timeout`. #1201 #1260
- Exit with error code when appropriate. #1238
- MSSQL connection accepts `host` as an alias for `server` in accordance with other dialects. #1239
- Add `query-response` event. #1231
- Correct behaviour of sibling nested transactions. #1226
- Support `RETURNING` with `UPDATE`. (Oracle) #1253
- Throwing callbacks from transactions automatically rolls them back. #1257
- Fixes to named `Raw` bindings. #1251
- `timestamps` accepts an argument to set `NOT NULL` and default to current timestamp.
- Add `TableBuilder#inherits` for PostgreSQL. #601
- Wrap index names. #1289
- Restore coffeescript knexfiles and configurations. #1292
- Add `andWhereBetween` and `andWhereNotBetween` #1132
- Fix `valueForUndefined` failure. #1269
- `renameColumn` no longer drops default value or nullability. #1326
- Correct MySQL2 error handling. #1315
- Fix MSSQL `createTableIfNotExists`. #1362
- Fix MSSQL URL parsing. #1342
- Update Lodash to 4.6.0 #1242
- Update Bluebird to 3.3.4 #1279

0.10.0 — *15 Feb, 2016*
Breaking Changes:

- `insert` and `update` now ignore `undefined` values. Back compatibility is provided through the option `useNullAsDefault`. #1174, #1043

Other Changes:

- Add `countDistinct`, `avgDistinct` and `sumDistinct`. #1046
- Add `schema.jsonb`. Deprecated `schema.json(column, true)`. #991
- Support binding identifiers with `??`. #1103
- Restore `query` event when triggered by transactions. #855
- Correct question mark escaping in rendered queries. #519, #1058
- Add per-dialect escaping, allowing quotes to be escaped correctly. #886, #1095
- Add MSSQL support. #1090
- Add migration locking. #1094
- Allow column aliases to contain `.`. #1181
- Add `batchInsert`. #1182
- Support non-array arguments to `knex.raw`.
- Global `query-error` event. #1163
- Add `batchInsert`. #1182
- Better support for Mysql2 dialect options. #980
- Support for `acquireConnectionTimeout` default 60 seconds preventing #1040 from happening. #1177
- Fixed constraint name escaping when dropping a constraint. #1177
- Show also `.raw` queries in debug output. #1169
- Support for `cli` to use basic configuration without specific environment set. #1101

0.9.0 — *Nov 2, 2015*

- Fix error when merging `knex.raw` instances without arguments. #853
- Fix error that caused the connection to time out while streaming. #849

- Correctly parse SSL query parameter for PostgreSQL. #852
- Pass `compress` option to MySQL2. #843
- Schema: Use `timestamp with timezone` by default for `time`, `datetime` and `timestamp` for Oracle. #876
- Add `QueryBuilder#modify` #881
- Add LiveScript and Early Gray support for seeds and migrations.
- Add `QueryBuilder#withSchema` #518
- Allow escaping of `?` in `knex.raw` queries. #946
- Allow `0` in join clause. #953
- Add migration config to allow disabling/enabling transactions per migration. #834

0.8.6 — *May 20, 2015*

Fix for several transaction / migration issues, #832, #833, #834, #835

0.8.5 — *May 14, 2015*

Pool should be initialized if no pool options are specified

0.8.4 — *May 13, 2015*

Pool should not be initialized if {max: 0} is sent in config options

0.8.3 — *May 2, 2015*

Alias postgresql -> postgres in connection config options

0.8.2 — *May 1, 2015*

Fix regression in using query string in connection config

0.8.1 — *May 1, 2015*

- Warn rather than error when implicit commits wipe out savepoints in mysql / mariadb, #805.
- Fix for incorrect seed config reference, #804

0.8.0 — *Apr 30, 2015*
New Features:

- Fixes several major outstanding bugs with the connection pool, switching to Pool2 in place of generic-pool-redux
- strong-oracle module support
- Nested transactions automatically become savepoints, with `commit` & `rollback` releasing or rolling back the current savepoint.
- Database seed file support, #391
- Improved support for sub-raw queries within raw statements
- Migrations are now wrapped in transactions where possible
- Subqueries supported in insert statements, #627
- Support for nested having, #572
- Support object syntax for joins, similar to "where" #743

Major Changes:

- Transactions are immediately invoked as A+ promises, #470 (this is a feature and should not actually break anything in practice)
- Heavy refactoring internal APIs (public APIs should not be affected)

"Other Changes:

- Allow mysql2 to use non-default port, #588
- Support creating & dropping extensions in PostgreSQL, #540
- CLI support for knexfiles that do not provide environment keys, #527
- Added sqlite3 dialect version of whereRaw/andWhereRaw (#477)

0.7.5 — *Mar 9, 2015*

Fix bug in validateMigrationList, (#697)

0.7.4 — *Feb 25, 2015*

- Fix incorrect order of query parameters when using subqueries, #704
- Properly handle limit 0, (#655)
- Apply promise args from then instead of explicitly passing.
- Respect union parameter as last argument (#660).
- Added sqlite3 dialect version of whereRaw/andWhereRaw (#477).
- Fix SQLite dropColumn doesn't work for last column (#544).
- Add POSIX operator support for Postgres (#562)
- Sample seed files now correctly (#391)

0.7.3 — *Oct 3, 2014*

- Support for `join(table, rawOrBuilder)` syntax.
- Fix for regression in PostgreSQL connection (#516)

0.7.2 — *Oct 1, 2014*

Fix for regression in migrations

0.7.1 — *Oct 1, 2014*

Better disconnect handling & pool removal for MySQL clients, #452

0.7.0 — *Oct 1, 2014*
New Features:

- Oracle support, #419
- Database seed file support, #391
- Improved support for sub-raw queries within raw statements

Breaking Changes:

- "collate nocase" no longer used by default in sqlite3 #396

Other Changes:

- Bumping Bluebird to ^2.x
- Transactions in websql are now a no-op (unsupported) #375
- Improved test suite
- knex.fn namespace as function helper (knex.fn.now), #372
- Better handling of disconnect errors
- Support for offset without limit, #446
- Chainable first method for mysql schema, #406
- Support for empty array in `whereIn`
- Create/drop schema for postgres, #511
- Inserting multiple rows with default values, #468
- Join columns are optional for cross-join, #508
- Flag for creating jsonb columns in Postgresql, #500

0.6.22 — *July 10, 2014*

Bug fix for properly binding postgresql streaming queries, (#363)

0.6.21 — *July 9, 2014*

- Bug fix for raw queries not being transaction context aware, (#351).
- Properly forward stream errors in sqlite3 runner, (#359)

0.6.20 — *June 30, 2014*

Allow case insensitive operators in sql clauses, (#344)

0.6.19 — *June 27, 2014*

- Add `groupByRaw` / `orderByRaw` methods, better support for raw statements in group / order (#282).
- Support more config options for node-mysql2 dialect (#341).
- CLI help text fix, (#342)

0.6.18 — *June 25, 2014*

Patch for the method, calling without a handler should return the stream, not a promise (#337)

0.6.17 — *June 23, 2014*

Adding missing map / reduce proxies to bluebird's implementation

0.6.16 — *June 18, 2014*

- Increment / decrement returns the number of affectedRows (#330).
- Allow --cwd option flag to be passed to CLI tool (#326)

0.6.15 — *June 14, 2014*

Added the as method for aliasing subqueries

0.6.14 — *June 14, 2014*

whereExists / whereNotExists may now take a query builder instance as well as a callback

0.6.13 — *June 12, 2014*

- Fix regression with onUpdate / onDelete in PostgreSQL, (#308).
- Add missing `Promise` require to knex.js, unit test for knex.destroy (#314)

0.6.12 — *June 10, 2014*

Fix for regression with boolean default types in PostgreSQL

0.6.11 — *June 10, 2014*

Fix for regression with queries containing multiple order by statements in sqlite3

0.6.10 — *June 10, 2014*

Fix for big regression in memoization of column names from 0.5 -> 0.6

0.6.9 — *June 9, 2014*

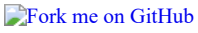
Fix for regression in specificType method

0.6.8 — *June 9, 2014*

Package.json fix for CLI

0.6.7 — *June 9, 2014*

- Adds support for node-mysql2 library.



- Bundles CLI with the knex install, various related migrate CLI fixes

0.6.6 — *June 9, 2014*

- console.warn rather than throw when adding foreignKeys in SQLite3.
- Add support for dropColumn in SQLite3.
- Document `raw.wrap`

0.6.5 — *June 9, 2014*

Add missing _ require to WebSQL builds

0.6.4 — *June 9, 2014*

Fix & document schema.raw method

0.6.3 — *June 6, 2014*

- Schema methods on transaction object are now transaction aware (#301).
- Fix for resolved value from transactions, (#298).
- Undefined columns are not added to builder

0.6.2 — *June 4, 2014*

- Fix regression in raw query output, (#297).
- Fix regression in "pluck" method (#296).
- Document first method

0.6.1 — *June 4, 2014*

Reverting to using .npmignore, the "files" syntax forgot the knex.js file

0.6.0 — *June 4, 2014*
Major Library refactor:

- Major internal overhaul to clean up the various dialect code.
- Improved unit test suite.
- Support for the mariasql driver.
- More consistent use of raw query bindings throughout the library.
- Queries are more composable, may be injected in various points throughout the builder.
- Added streaming interface
- Deprecated 5 argument join in favor of additional join methods.
- The wrapValue function to allow for array column operations in PostgreSQL (#287).
- An explicit connection can be passed for any query (#56).
- Drop column support for sqlite3
- All schema actions are run sequentially on the same connection if chained.
- Schema actions can now be wrapped in a transaction
- `.references(tableName.columnName)` as shorthand for `.references(columnName).inTable(tableName)`
- `.join('table.column', 'otherTable.column')` as shorthand for .join('table.column', '=', 'otherTable.column')
- Streams are supported for selects, passing through to the streaming capabilities of node-mysql and node-postgres
- For More information, see this pull-request

0.5.15 — *June 4, 2014*

Dropped indexes feature now functions correctly, (#278)

0.5.14 — *May 6, 2014*

Remove the charset encoding if it's utf8 for mysql, as it's the default but also currently causes some issues in recent versions of node-mysql

0.5.13 — *April 2, 2014*

Fix regression in array bindings for postgresql (#228)

0.5.12 — *Mar 31, 2014*

Add more operators for where clauses, including && (#226)

0.5.11 — *Mar 25, 2014*

- `.where(col, 'is', null)` or `.where(col, 'is not', null)` are not supported (#221).
- Case insensitive `where` operators now allowed (#212).
- Fix bug in increment/decrement truncating to an integer (#210).
- Disconnected connections are now properly handled & removed from the pool (#206).
- Internal tweaks to binding concatenations for performance (#207)

0.5.10 — *Mar 19, 2014*

Add the .exec method to the internal promise shim

0.5.9 — *Mar 18, 2014*

Remove error'ed connections from the connection pool (#206), added support for node-postgres-pure (pg.js) (#200)

0.5.8 — *Feb 27, 2014*

Fix for chaining on forUpdate / forShare, adding map & reduce from bluebird

0.5.7 — *Feb 18, 2014*

Fix for a null limit / offset breaking query chain (#182)

0.5.6 — *Feb 5, 2014*

Bump bluebird dependency to ~1.0.0, fixing regression in Bluebird 1.0.2 (#176)

0.5.5 — *Jan 28, 2014*

- Fix for the exit code on the migrations cli (#151).
- The `init` method in `knex.migrate` now uses `this.config` if one isn't passed in (#156)

0.5.4 — *Jan 7, 2014*

Fix for using raw statements in defaultTo schema builder methods (#146)

0.5.3 — *Jan 2, 2014*

Fix for incorrectly formed sql when aggregates are used with columns (#144)

0.5.2 — *Dec 18, 2013*

Adding passthrough "catch", "finally" to bluebird implementations, use bluebird's "nodeify" internally for exec

0.5.1 — *Dec 12, 2013*

- The returning in PostgreSQL may now accept * or an array of columns to return. If either of these are passed, the response will be an array of objects rather than an array of values. Updates may also now use a `returning` value. (#132)
- Added `bigint` and `bigserial` type to PostgreSQL. (#111)
- Fix for the specificType schema call (#118)
- Several fixes for migrations, including migration file path fixes, passing a Promise constructor to the migration `up` and `down` methods, allowing the "knex" module to be used globally, file ordering on migrations, and other small improvements. (#112-115, #125, #135)

0.5.0 — *Nov 25, 2013*

- Initial pass at a migration api.
- Aggregate methods are no longer aliased as "aggregate", but may now be aliased and have more than one aggregate in a query (#108, #110).
- Adding bigint and bigserial to PostgreSQL (#111).
- Bugfix on increment/decrement values (#100).
- Bugfix with having method (#107).
- Switched from when.js to bluebird for promise implementation, with shim for backward compatibility.
- Switched from underscore to lodash, for semver reliability

0.4.13 — *Oct 31, 2013*

Fix for aggregate methods on toString and clone, (#98)

0.4.12 — *Oct 29, 2013*

Fix incorrect values passed to float in MySQL and decimal in PostgreSQL

0.4.11 — *Oct 15, 2013*

Fix potential sql injection vulnerability in orderBy, thanks to @sebgie

0.4.10 — *Oct 14, 2013*

- Added forUpdate and forShare for select modes in transactions. (#84)
- Fix bug where current query chain type is not copied on clone. (#90)
- Charset and collate are now added as methods on the schema builder. (#89)
- Added `into` as an alias of from, for builder syntax of: `insert(value).into(tableName)`
- Internal pool fixes. (#90)

0.4.9 — *Oct 7, 2013*

- Fix for documentation of hasColumn, ensure that `hasColumn` works with MySQL (#87).
- More cleanup of error messages, showing the original error message concatenated with the sql and bindings

0.4.8 — *Oct 2, 2013*

Connections are no longer pushed back into the pool if they never existed to begin with (#85)

0.4.7 — *Sep 27, 2013*

The column is now a documented method on the builder api, and takes either an individual column or an array of columns to select

0.4.6 — *Sep 25, 2013*

Standardizing handling of errors for easier debugging, as noted in (#39)

0.4.5 — *Sep 24, 2013*

Fix for hasTable always returning true in MySQL (#82), fix where sql queries were duplicated with multiple calls on toSql with the schema builder

0.4.4 — *Sep 22, 2013*

Fix for debug method not properly debugging individual queries

0.4.3 — *Sep 18, 2013*

Fix for underscore not being defined in various grammar files

0.4.2 — *Sep 17, 2013*

Fix for an error being thrown when an initialized ClientBase instance was passed into Knex.initialize. pool.destroy now optionally accepts a callback to notify when it has completed draining and destroying all connections

0.4.1 — *Sep 16, 2013*

Cleanup from the 0.4.0 release, fix a potential exploit in "where" clauses pointed out by Andri Möll, fix for clients not being properly released from the pool #70, fix for where("foo", "<>", null) doing an "IS NULL" statement

0.4.0 — *Sep 13, 2013*
Breaking Changes:

- Global state is no longer stored in the library, an instance is returned from `Knex.initialize`, so you will need to call this once and then reference this `knex` client elsewhere in your application.
- Lowercasing of `knex.raw`, `knex.transaction`, and `knex.schema`.
- Created columns are now nullable by default, unless `notNullable` is chained as an option.
- Keys created with `increments` are now assumed to be unsigned (MySQL) by default.
- The `destroyAllNow` is no longer called by the library on `process.exit` event. If you need to call it explicitly yourself, you may use `knex.client.destroyPool`

0.2.6 — *Aug 29, 2013*

Reject the transaction promise if the transaction "commit" fails, (#50)

0.2.5 — *Aug 25, 2013*

Fix error if a callback isn't specified for exec, (#49)

0.2.4 — *Aug 22, 2013*

Fix SQLite3 delete not returning affected row count, (#45)

0.2.3 — *Aug 22, 2013*

Fix insert with default values in PostgreSQL and SQLite3, (#44)

0.2.2 — *Aug 20, 2013*

Allowing Raw queries to be passed as the primary table names

0.2.1 — *Aug 13, 2013*

Fix for an array passed to insert being mutated

0.2.0 — *Aug 7, 2013*
Breaking changes:

- hasTable now returns a boolean rather than a failed promise.
- Changed syntax for insert in postgresql, where the `id` is not assumed on inserts (#18). The second parameter of insert is now required to return an array of insert id's for the last insert.
- The timestamp method on the schema builder now uses a `dateTime` rather than a `timestamp`

0.1.8 — *July 7, 2013*

Somehow missing the != operator. Using _.find rather than _.where in getCommandsByName(#22)

0.1.7 — *June 12, 2013*

Ensures unhandled errors in the exec callback interface are re-thrown

0.1.6 — *June 9, 2013*

Renaming beforeCreate to afterCreate. Better handling of errors in the connection pooling

0.1.5 — *June 9, 2013*

Added the ability to specify beforeCreate and beforeDestroy hooks on the initialize's options.pool to perform any necessary database setup/teardown on connections before use (#14). where and having may now accept Knex.Raw instances, for consistency (#15). Added an orHaving method to the builder. The ability to specify bindings on Raw queries has been removed

0.1.4 — *May 22, 2013*

defaultTo now accepts "false" for boolean columns, allows for empty strings as default values

0.1.3 — *May 18, 2013*

Enabling table aliases (#11). Fix for issues with transactions not functioning (#12)

0.1.2 — *May 15, 2013*

Bug fixes for groupBy (#7). Mysql using collation, charset config settings in createTable. Added engine on schemaBuilder specifier (#6). Other doc fixes, tests

0.1.1 — *May 14, 2013*

Bug fixes for sub-queries, minor changes to initializing "main" instance, adding "pg" as a valid parameter for the client name in the connection settings

0.1.0 — *May 13, 2013*

Initial Knex release