
epam
SQL DATABASES IN NODE.JS
Andrii Kochkin
JUNE 18, 2018

ABOUT ME

ANDRII KOCHKIN

Senior Software Engineer

Andrii_Kochkin@epam.com

Development experience:

* 15+ years of development experience in Software Engineering with full-stack technical expertise;

* 5+ years of commercial experience as a JavaScript Developer;

Key Developer on CCC-ARCH project

Main Focus: Front-End and Backend Development

AGENDA OF THE LECTURE

- Introduction to Databases;
- Understanding Data Modeling;
- Overview of SQL possibilities;
- Native DB driver;
- ORM concepts;
- Sequelize ORM driver.


INTRODUCING TO DATABASES

DEFINITION OF A RELATION DATABASE

A relational database is a collection of relations or two-dimensional tables.

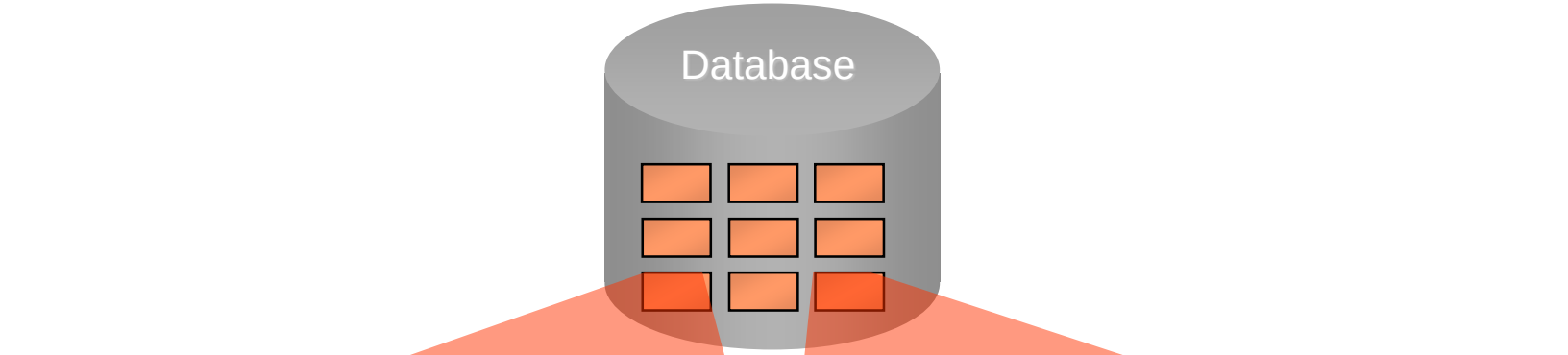


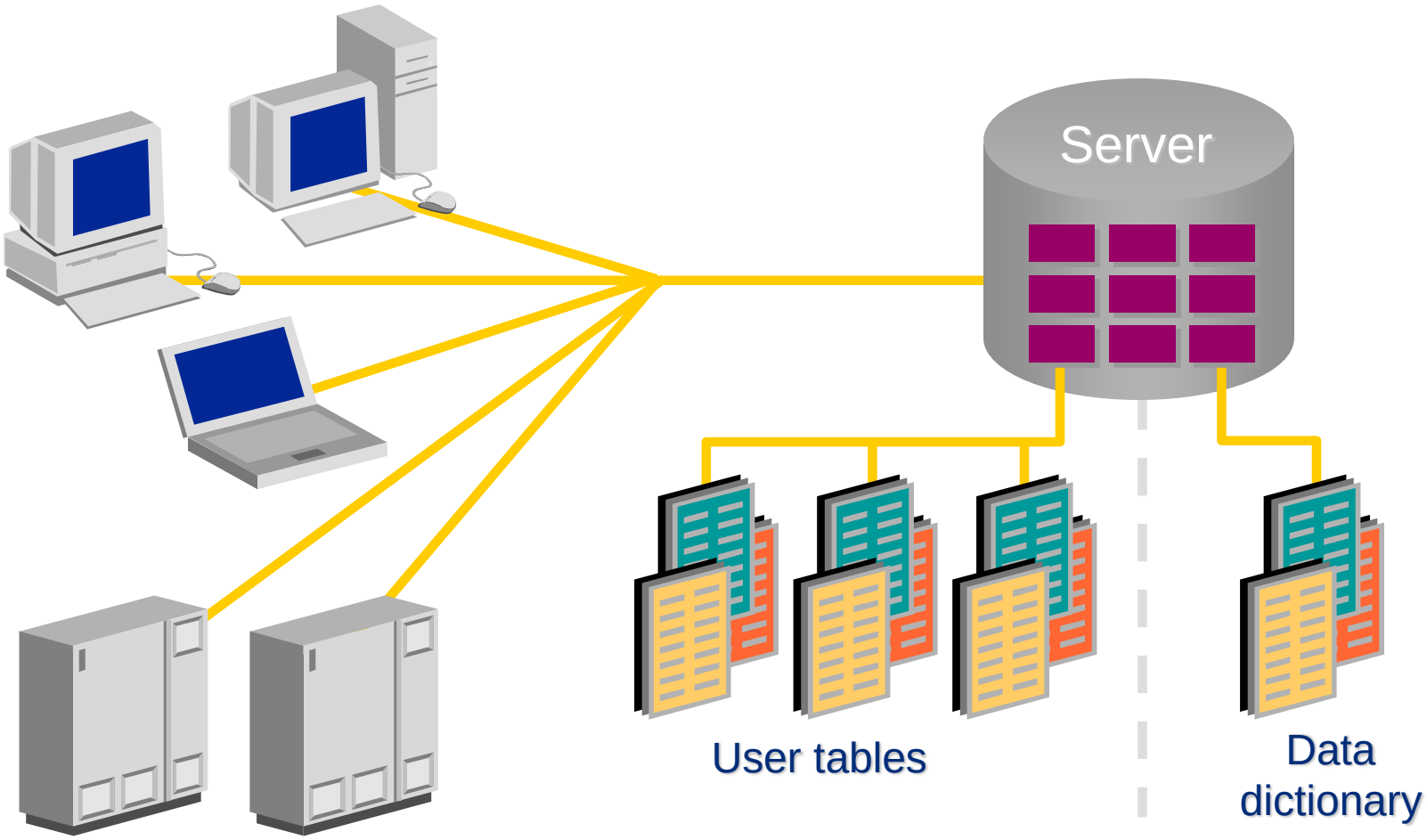
Table name: EMP

EMPNO	ENAME	JOB	DEPTNO
7839	KING	PRESIDENT	10
7698	BLAKE	MANAGER	30
7782	CLARK	MANAGER	10
7566	JONES	MANAGER	20

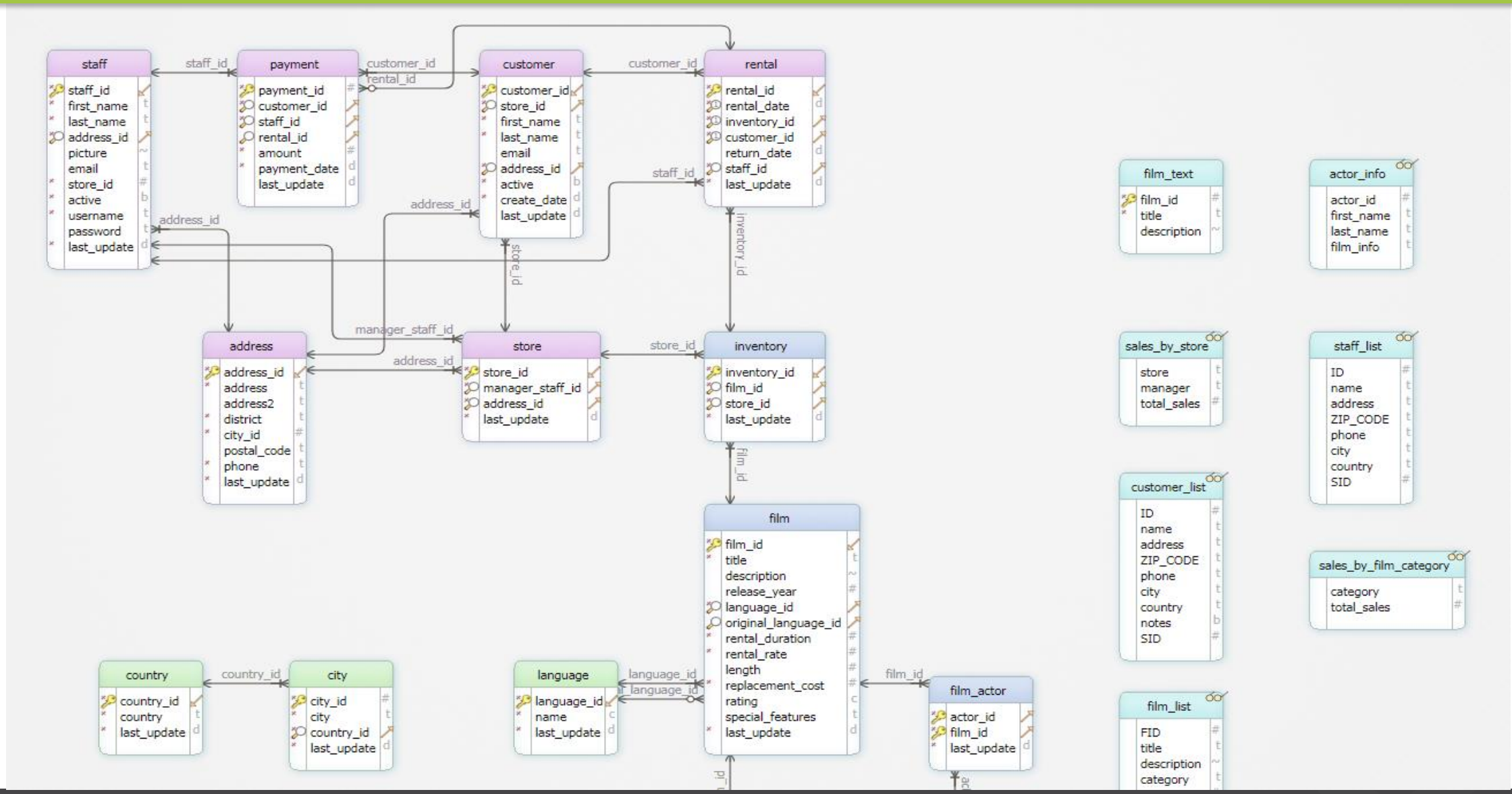
Table name: DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

RELATION DATABASE MANAGEMENT SYSTEM (RDBMS)

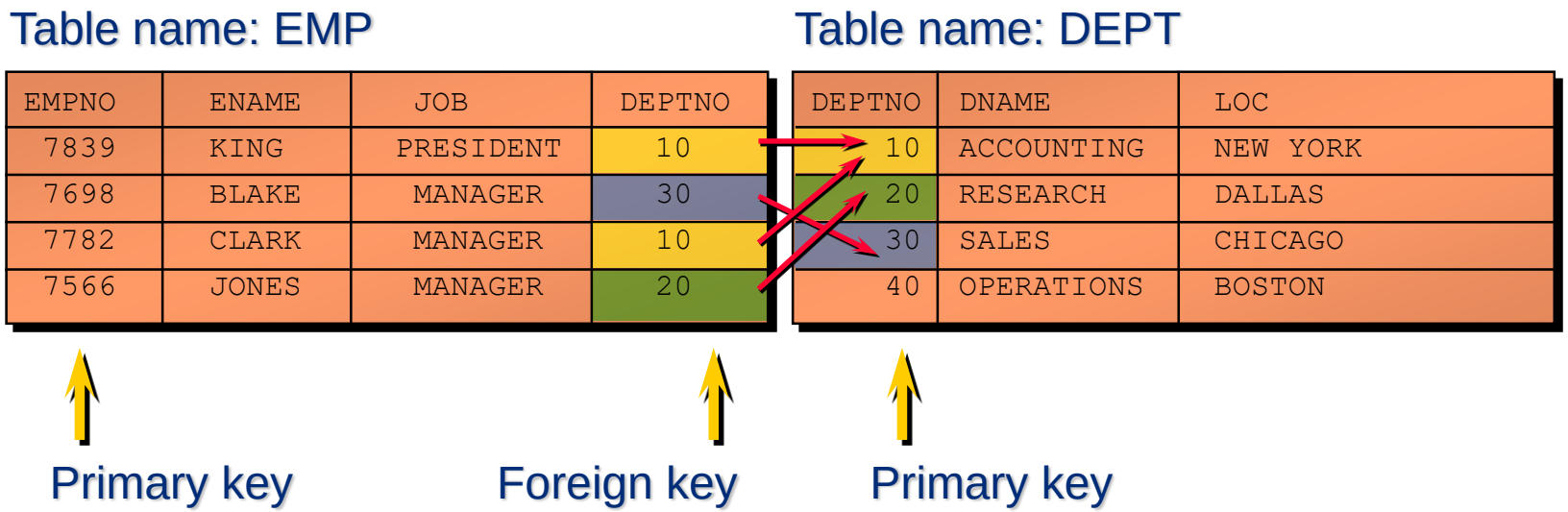


SQL DATA STRUCTURE



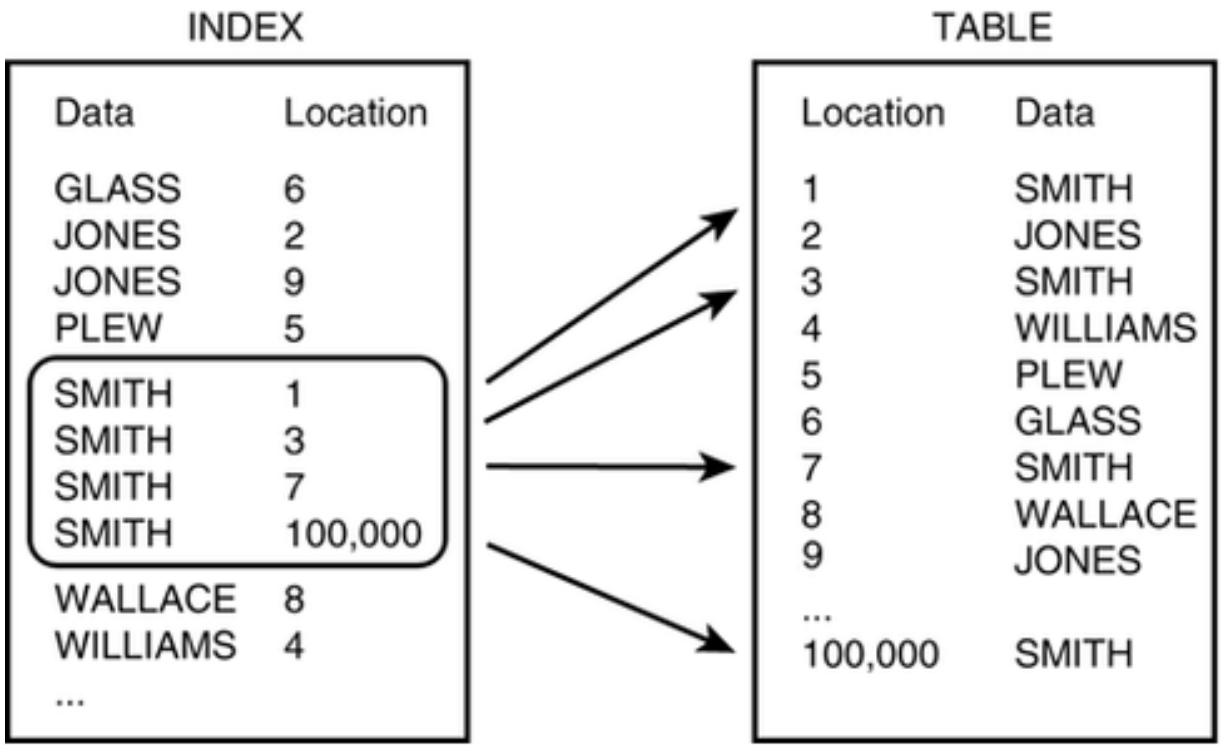
RELATING MULTIPLE TABLES

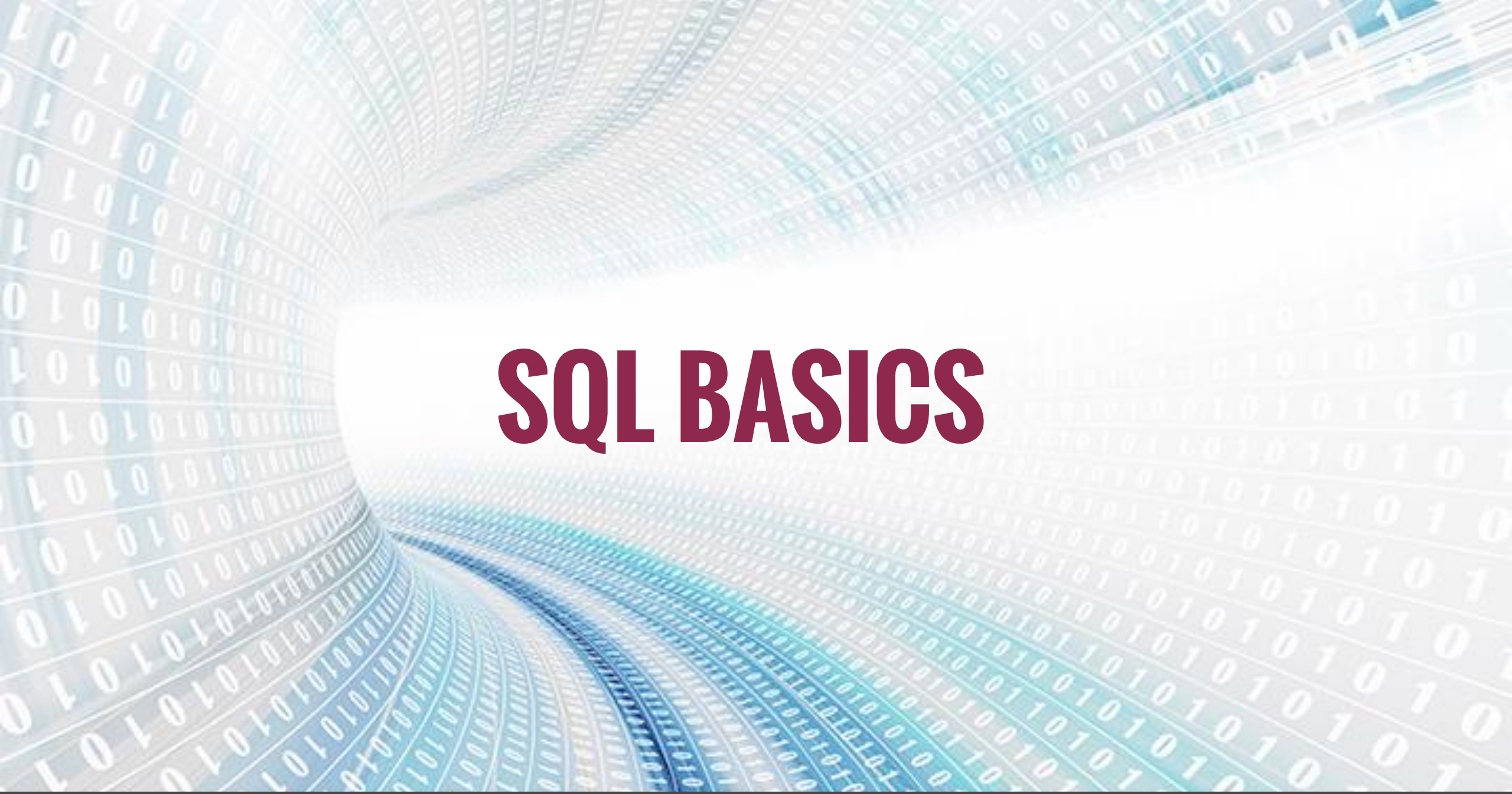
- Each row of data in a table is uniquely identified by a primary key (PK).
- You can logically relate data from multiple tables using foreign keys (FK).



INDEXING

When to Create an index




SQL BASICS

What is SQL?

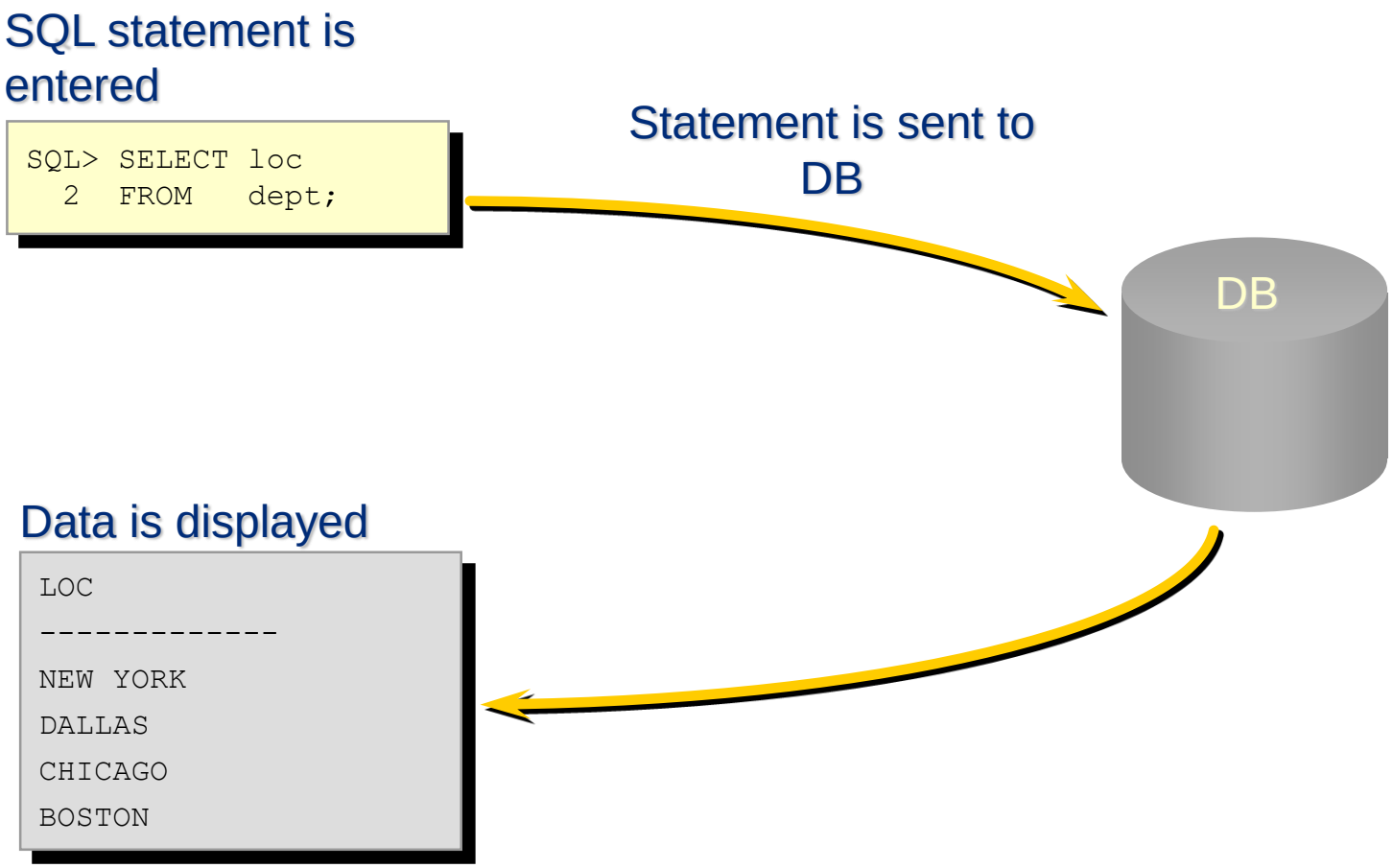
SQL is an abbreviation for "Structured Query Language".

SQL is a language used to build database applications that need to query relational databases.

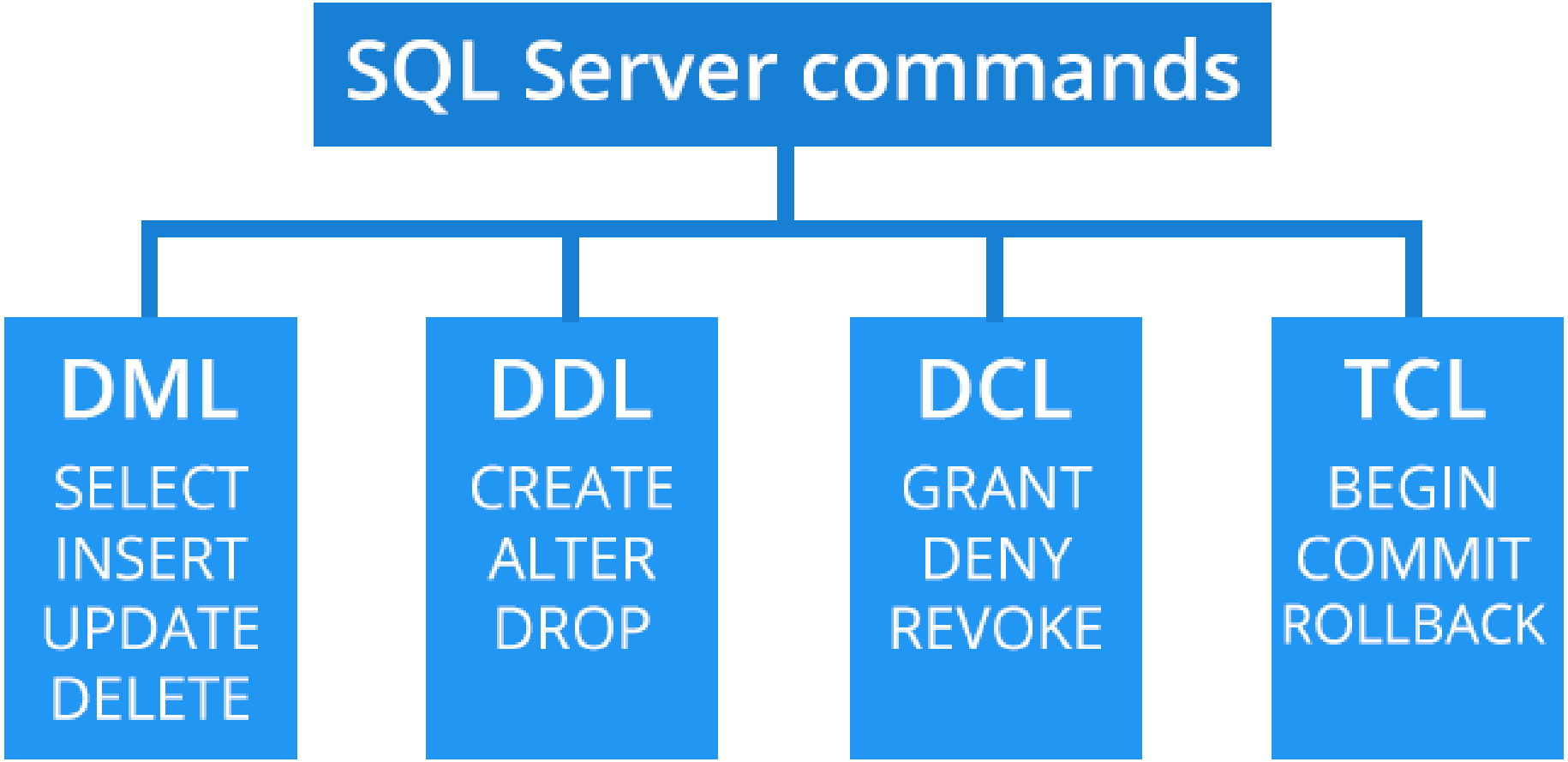
SQL has statements such as CREATE, SELECT, INSERT, UPDATE, DELETE, etc., just like there are statements such as assignment statement, if statement, while statement, etc., in general purpose programming languages such as C, C++ and Java.

SQL is a language for databases just like C, C++ and Java are languages for general purpose programming.

COMMUNICATING WITH A RDBMS USING SQL



SQL COMMANDS



SQL SELECT

Syntax

```
SELECT [DISTINCT|ALL]{*|column|column_expression [AS new_name][,…]}
   FROM   table_name [alias] [, … ]
     [WHERE  condition]
     [GROUP BY  column_list]
     [HAVING  condition]
     [ORDER BY  column_list [ASC|DESC]];
```

- *column* represents a column name.
- *column_expression* represents an expression on a column.
- *table_name* is the name of an existing database table or view.
- FROM specifies the table(s) to be used.
- WHERE filters the rows subject to some condition.
- GROUP BY forms groups of rows with the same column name.
- SELECT specifies which column are to appear in the output.
- ORDER BY specifies the order of the output.
- Order of the clauses in the SELECT statement can not be changed.
- The result of a query is another table.

SQL SELECT EXAMPLES

```
SELECT  sno, fname, lname, position, sex, dob, salary, bno
  FROM  staff;
```

OR

```
SELECT  *
  FROM  staff;
```

Sno	FName	LName	position	Sex	DOB	Salary	bno
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003

```
SELECT sno, fname, lname, salary
  FROM staff
    WHERE  salary > 10000;
```

Sno	FName	LName	Salary
SL21	John	White	30000
SG37	Ann	Beech	12000
SG14	David	Ford	18000
SG5	Susan	Brand	24000

```
SELECT propertyNo, type, rooms, rent
  FROM property
     ORDER BY  type, rent  DESC;
```

PropertNo	Type	Rooms	Rent
PG16	Flat	4	450
PL94	Flat	4	400
PG36	Flat	3	370
PG4	House	3	650
PA14	House	6	600

SQL INSERT

Syntax

```
INSERT INTO table_name [(column (,…))]
    { VALUES (date_value (,…)) |  subquery };
```

- *table_name* may be either a base table or an updatable view.
- *column_list* represents a list of one or more column names separated by commas.
- If omitted, SQL assumes a list of all columns in their original CREATE TABLE order.
- If specified, then any columns that are omitted from the list must have been declared as NULL column.
- *data_value* must match the *column_list* as follows:
  - The number of items in each list must be same.
  - There must be a direct correspondence in the position of items in the two lists, so that the first item in the *data_value_list* applies to the first item in the *column_list*, and so on.
  - The data type of each item in the *data_value_list* must be compatible with the data type of the corresponding column.

SQL INSERT EXAMPLE

Example:
Insert a new row into the staff table supplying data for all mandatory columns, knowing that the sex and birth date are optional fields.

```
INSERT INTO staff (Sno, fname, lname, position, salary, bno)
VALUES ('SG16', 'Alan', 'Brown', 'Assistant', 8300, 'B003');
```

Alternative:

```
INSERT INTO staff
VALUES ('SG16', 'Alan', 'Brown', 'Assistant', NULL, NULL, 8300,
        'B003');
```

SQL UPDATE

Syntax

```
    UPDATE table_name
       SET column_name1 = data_value1 [, column_namei =
data_valuei ...]
            [WHERE  search_condition]
```

- *table_name* may be either a base table or an updatable view.
- The SET clause specifies the names of one or more columns that are updated for all rows in the table.
- Only rows that satisfy the *search_condition* are updated.
- *data_values* must be compatible with the data types for the corresponding columns.

SQL UPDATE EXAMPLE

STAFF(sno, fname, lname, position, sex, DOB, salary, bno)

Example:
Give all staff a 3% pay increase.

```
UPDATE  staff
  SET  salary = salary * 1.03;
```

Example:
Give all managers a 3% pay increase.

```
UPDATE  staff
  SET  salary = salary * 1.03
    WHERE  position = 'Manager';
```

SQL DELETE

Syntax

```
DELETE  FROM  table_name
        [WHERE  search_condition];
```

- *table_name* may be either a base table or an updatable view.
- Only rows that satisfy the *search_condition* are deleted.
- If no *search_condition* is omitted, all rows are deleted from the table.
- DELETE does not delete the table itself, only rows in the table.

SQL DELETE EXAMPLE

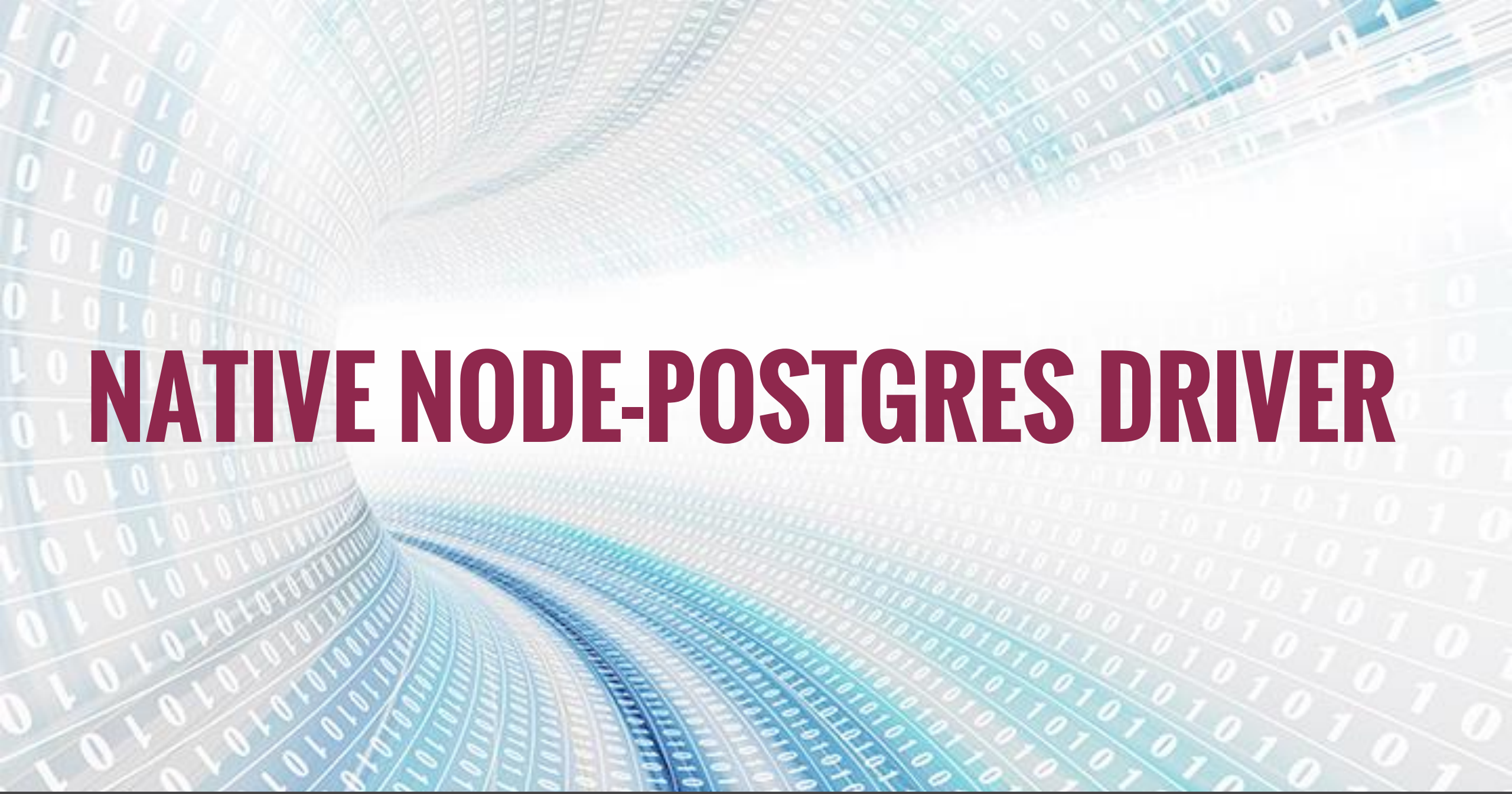
STAFF(sno, fname, lname, position, sex, DOB, salary, bno)

Example:
Delete all staff in branch B003.

```
DELETE FROM  staff
  WHERE  bno = 'B003';
```

Example:
Delete all staff.

```
DELETE FROM staff;
```


NATIVE NODE-POSTGRES DRIVER

SETTING UP A CONNECTION

Getting started

This is the simplest possible way to connect, query, and disconnect with async/await:

```
const { Client } = require('pg')
const client = new Client()

await client.connect()

const res = await client.query('SELECT $1::text as message', ['Hello world!'])
console.log(res.rows[0].message) // Hello world!
await client.end()
```

SETTING UP A CONNECTION

Getting started

And here's the same thing with callbacks:

```
var Pg = require('pg').Client;
var conString = "postgres://postgres:123456@domain.com/nodejs";

var pg = new Pg(conString);
pg.connect();

var query = pg.query("select * from node.users order by id", function(err, result){
    var json = JSON.stringify(result.rows);
    console.log(json);
});
```

QUERIES

Parameterized query

With callback and promise

```
const text = 'INSERT INTO users(name, email) VALUES($1, $2) RETURNING *'
const values = ['brianc', 'brian.m.carlson@gmail.com']

client.query(text, values, (err, res) => {
    if (err) {
        console.log(err.stack)
    } else {
        console.log(res.rows[0])
        // { name: 'brianc', email: 'brian.m.carlson@gmail.com' }
    }
})

client.query(text, values)
    .then(res => {
        console.log(res.rows[0])
        // { name: 'brianc', email: 'brian.m.carlson@gmail.com' }
    })
    .catch(e => console.error(e.stack))
```

QUERIES

Row mode with callback and promise

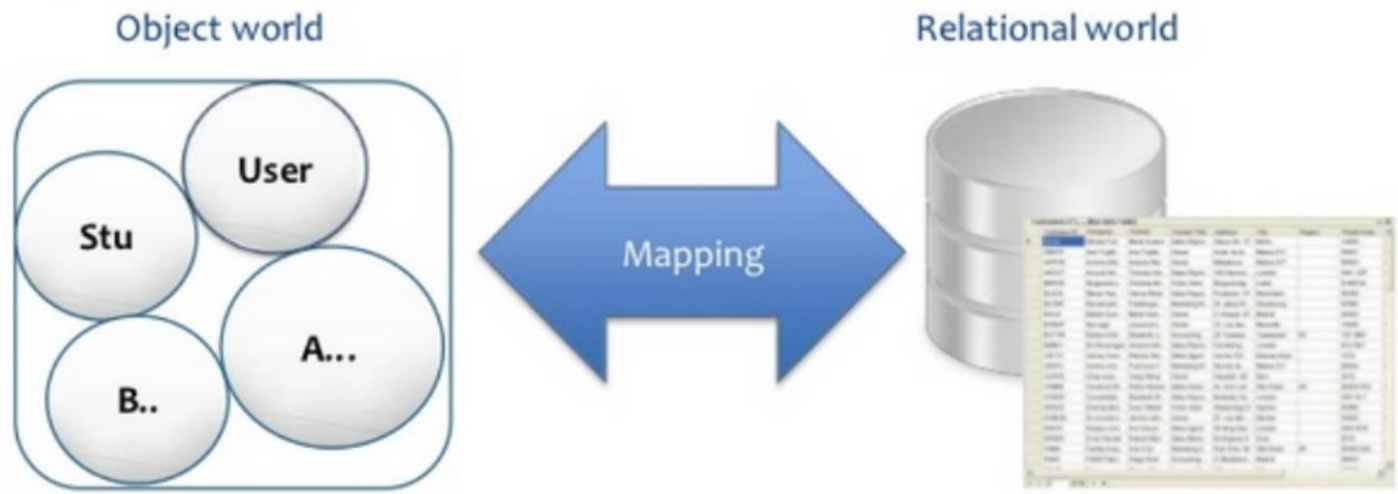
```
const query = {
    text: 'SELECT $1::text as first_name, select $2::text as last_name',
    values: ['Brian', 'Carlson'],
    rowMode: 'array',
};

client.query(query, (err, res) => {
    if (err) {
        console.log(err.stack)
    } else {
        console.log(res.fields.map(f => field.name)) // ['first_name', 'last_name']
        console.log(res.rows[0]) // ['Brian', 'Carlson']
    }
})

client.query(query)
    .then(res => {
        console.log(res.fields.map(f => field.name)) // ['first_name', 'last_name']
        console.log(res.rows[0]) // ['Brian', 'Carlson']
    })
    .catch(e => console.error(e.stack))
```


ORM: WHAT IS THIS?

ORM

What is ORM?

- Relational Database
- Objects
- Mapping between Objects & Relational Database



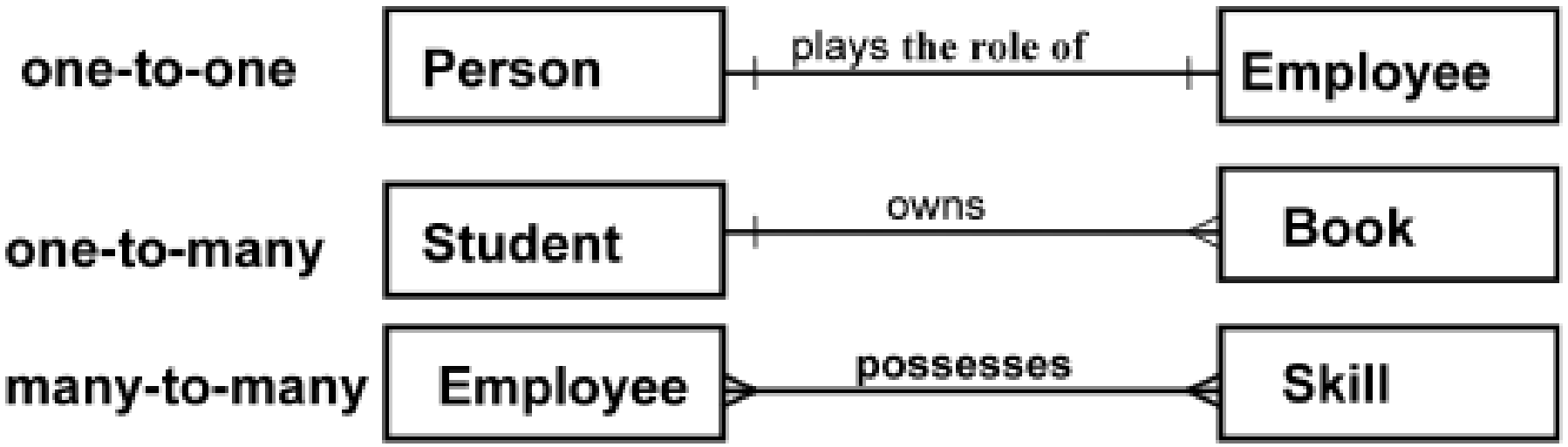
Why ORM?

- Leverages existing knowledge and skills related to OOP
- Level of abstraction
- Saves programmer time

CRUD

1. Create
   - User.create(:name => ‘asd’, :email => ‘asd@wer.com’)
2. Read
   - @user = User.find(1)
3. Update
   - @user.name = ‘new name’
   - @user.save
4. Delete
   - @user.delete

ASSOCIATIONS



Advantages of ORM

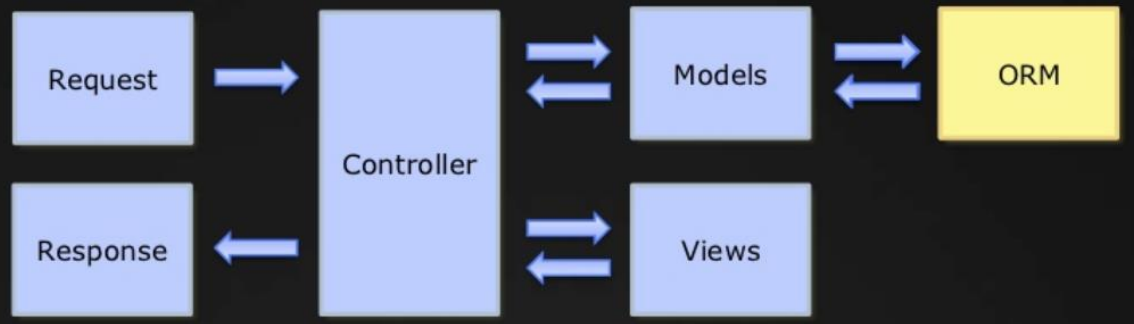
- ORM frees the programmer from dealing with simple repetitive database queries
- Automatically mapping the database to business objects lets programmers focus more on business problems and less with data storage
- The mapping process can aid in data verification and security before reaching the database
- ORM can provide a caching layer above the database

Disadvantages of ORM

- ORM adds a least one small layer of application code, potentially increasing processing overhead.
- ORM can load related business objects on demand.
- An ORM system's design might dictate certain database design decisions.

ORM vs MVC

Using an ORM in an MVC framework



- The ORM is your data abstraction layer
- The Model contains your business logic
- What not to do:
  - have the business logic in the controller and use the ORM as your model
- Some ORM's allow you to merge the functionality of the Model and the ORM
  - but think carefully about the complexity…

SEQUELIZE ORM DRIVER

SEQUELIZE

Sequelize is a promise-based ORM for Node.js and io.js. It supports PostgreSQL, MySQL, MariaDB, SQLite and MSSQL and features transaction support, relations, read replication and more. Starting from 4.0.0 Sequelize will only support Node v4 and above to use ES6 features.

SETTING UP A CONNECTION

Getting started

you can simply use a connection uri

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname');
```

SETTING UP A CONNECTION

Getting started

Setup with parameters

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('database', 'username', 'password', {
    host: 'localhost',
    dialect: 'mysql'|'sqlite'|'postgres'|'mssql',

    pool: {
        max: 5,
        min: 0,
        idle: 10000
    },

    // SQLite only
    storage: 'path/to/database.sqlite'
});
```

TEST CONNECTION

You can use the .authenticate() function like this to test the connection.

```
sequelize
    .authenticate()
    .then(() => {
        console.log('Connection has been established successfully.');
    })
    .catch(err => {
        console.error('Unable to connect to the database:', err);
    });
```

MODEL DEFINITION

To define mappings between a model and a table, use the define method.

```
const Project = sequelize.define('project', {
    title: Sequelize.STRING,
    description: Sequelize.TEXT
});

const Task = sequelize.define('task', {
    title: Sequelize.STRING,
    description: Sequelize.TEXT,
    deadline: Sequelize.DATE
});
```

GETTERS & SETTERS

It is possible to define 'object-property' getters and setter functions on your models, these can be used both for 'protecting' properties that map to database fields and for defining 'pseudo' properties.

```
const Employee = sequelize.define('employee', {
    name: {
        type: Sequelize.STRING,
        allowNull: false,
        get() {
            const title = this.getDataValue('title');
            // 'this' allows you to access attributes of the instance
            return this.getDataValue('name') + ' (' + title + ')';
        },
    },
    title: {
        type: Sequelize.STRING,
        allowNull: false,
        set(val) {
            this.setDataValue('title', val.toUpperCase());
        }
    }
});
```

DATA TYPES

Below are some of the datatypes supported by sequelize:

```
Sequelize.STRING                      // VARCHAR(255)
Sequelize.TEXT                        // TEXT
Sequelize.INTEGER                     // INTEGER
Sequelize.BIGINT                      // BIGINT
Sequelize.FLOAT                       // FLOAT
Sequelize.REAL                        // REAL        PostgreSQL only.
Sequelize.DOUBLE                      // DOUBLE
Sequelize.DECIMAL                     // DECIMAL
Sequelize.DATE                        // DATETIME for mysql/sqlite,TIMESTAMP WITH TIME ZONE for postgres
Sequelize.DATEONLY                    // DATE without time.
Sequelize.BOOLEAN                     // TINYINT(1)
Sequelize.ENUM('value 1', 'value 2')  // An ENUM with allowed values 'value 1' and 'value 2'
Sequelize.ARRAY(Sequelize.TEXT)       // Defines an array. PostgreSQL only.
Sequelize.JSON                        // JSON column. PostgreSQL, SQLite and MySQL only.
Sequelize.JSONB                       // JSONB column. PostgreSQL only.
Sequelize.BLOB                        // BLOB (bytea for PostgreSQL)
Sequelize.UUID                        // UUID datatype for PostgreSQL and SQLite, CHAR(36)
```

VALIDATIONS

Model validations allow you to specify format/content/inheritance validations for each attribute of the model.

It runs on create, update and save. You can also call validate() to manually validate an instance

```
const Pub = Sequelize.define('pub', {
    name: { type: Sequelize.STRING },
    address: { type: Sequelize.STRING },
    latitude: {
        type: Sequelize.INTEGER,
        allowNull: true,
        defaultValue: null,
        validate: { min: -90, max: 90 }
    },
    longitude: {
        type: Sequelize.INTEGER,
        allowNull: true,
        defaultValue: null,
        validate: { min: -180, max: 180 }
    },
}, {
    validate: {
        bothCoordsOrNone() {
            if ((this.latitude === null) !== (this.longitude === null)) {
                throw new Error('Require either both latitude and longitude or neither')
            }
        }
    }
})
```

MODEL USAGE

find - Search for one specific element in the database

```
// search for known ids
Project.findById(123).then(project => {
    // project will be an instance of Project and stores the content of the table entry
    // with id 123. if such an entry is not defined you will get null
})

// search for attributes
Project.findOne({ where: {title: 'aProject'} }).then(project => {
    // project will be the first entry of the Projects table with the title 'aProject' || null
})

Project.findOne({
    where: {title: 'aProject'},
    attributes: ['id', ['name', 'title']]
}).then(project => {
    // project will be the first entry of the Projects table with the title 'aProject' || null
    // project.title will contain the name of the project
})

// limit the results of the query
Project.findAll({ limit: 10 })
```

MODEL USAGE

- **findOrCreate -** Search for a specific element or create it if not available
- **findAndCountAll -** Search for multiple elements in the database, returns both data and total count
- **findAll** - Search for multiple elements in the database

MODEL USAGE

Manipulating the dataset with limit, offset, order and group

To get more relevant data, you can use limit, offset, order and grouping:

```
// limit the results of the query
Project.findAll({ limit: 10 })

// step over the first 10 elements
Project.findAll({ offset: 10 })

// step over the first 10 elements, and take 2
Project.findAll({ offset: 10, limit: 2 })
```

QUERYING

Attributes:

list all the attributes of the model if you only want to add an aggregation

```
// This is a tiresome way of getting the number of hats...
Model.findAll({
    attributes: ['id', 'foo', 'bar', 'baz', 'quz', [sequelize.fn('COUNT', sequelize.col('hats')), 'no_hats']]
});

// This is shorter, and less error prone because it still works if you add / remove attributes
Model.findAll({
    attributes: { include: [[sequelize.fn('COUNT', sequelize.col('hats')), 'no_hats']] }
});

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

QUERYING

- Attributes
- Where
  - Basics
  - Operators
    - Range Operators
    - Combinations
    - Operators Aliases
    - Operators security
  - JSONB
    - Nested object
    - Nested key
    - Containment
  - Relations / Associations
- Pagination / Limiting
- Ordering

```
where: { authorId: 2 }
[Op.or]: [{a: 5}, {a: 6}] // (a = 5 OR a = 6)
[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15
rank: { [Op.or]: { [Op.lt]: 1000, [Op.eq]: null } }
const operatorsAliases = { $gt: Op.gt }  // $gt: 6
operatorsAliases: { $and: Op.and }
```

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

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

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

```
where: { state: Sequelize.col('project.state')}

Project.findAll({ offset: 5, limit: 5 })

order: ['title', 'DESC']
```

ASSOCIATIONS

One-To-One associations

```
const Player = this.sequelize.define('player', {/* attributes */});
const Team  = this.sequelize.define('team', {/* attributes */});

Player.belongsTo(Team); // Will add a teamId attribute to Player to hold the primary key value for Team
```

```
const User = this.sequelize.define('user', {/* attributes */})
const UserRole  = this.sequelize.define('userRole', {/* attributes */});

User.belongsTo(UserRole, {as: 'role'}); // Adds roleId to user rather than userRoleId
```

ASSOCIATIONS

One-To-Many associations

```
const User = sequelize.define('user', {/* ... */})
const Project = sequelize.define('project', {/* ... */})

// OK. Now things get more complicated (not really visible to the user :)).
// First let's define a hasMany association
Project.hasMany(User, {as: 'Workers'})

const City = sequelize.define('city', { countryCode: Sequelize.STRING });
const Country = sequelize.define('country', { isoCode: Sequelize.STRING });

// Here we can connect countries and cities base on country code
Country.hasMany(City, {foreignKey: 'countryCode', sourceKey: 'isoCode'});
City.belongsTo(Country, {foreignKey: 'countryCode', targetKey: 'isoCode'});
```

Managed transaction (auto-callback)

```
return sequelize.transaction(function (t) {

    // chain all your queries here. make sure you return them.
    return User.create({
        firstName: 'Abraham',
        lastName: 'Lincoln'
    }, {transaction: t}).then(function (user) {
        return user.setShooter({
            firstName: 'John',
            lastName: 'Boothe'
        }, {transaction: t});
    });

}).then(function (result) {
    // Transaction has been committed
    // result is whatever the result of the promise chain returned to the transaction callback
}).catch(function (err) {
    // Transaction has been rolled back
    // err is whatever rejected the promise chain returned to the transaction callback
});
```

Concurrent/Partial transactions

```
sequelize.transaction(function (t1) {
    return sequelize.transaction(function (t2) {
        // Pass in the `transaction` option to define/alter the transaction they belong to.
        return Promise.all([
            User.create({ name: 'Bob' }, { transaction: null }),
            User.create({ name: 'Mallory' }, { transaction: t1 }),
            User.create({ name: 'John' }) // this would default to t2
        ]);
    });
});
```

HOOKS

Hooks (also known as lifecycle events), are functions which are called before and after calls in sequelize are executed

```
User.hook('beforeValidate', (user, options) => {
            user.mood = 'happy';
 });

 User.addHook('afterValidate', 'someCustomName', (user, options) => {
            return sequelize.Promise.reject(new Error("I'm afraid I can't let you do that!"));
 });
```

Only a hook with name param can be removed:

```
Book.addHook('afterCreate', 'notifyUsers', (book, options) => {
  // ...
});

Book.removeHook('afterCreate', 'notifyUsers');
```

CONCLUSION

FAQ

F. A. Q. - ?

THANKS!

Andrii Kochkin

JUNE 18, 2018