There are a few ways to go through this exercise. If you had previous experience with databases and data modeling, you might want to experiment with other ways of solving this problem. For example, you could start with building an abstraction over your tables (repositories), which is a common way of interacting with databases.

However, in this solution, we will focus on interacting with SQL with minimal abstractions while still arriving at a solution that is relatively easy to read and maintain.

1. Familiarize yourself with the problem and database schema.

In the `README.md` file of this exercise, we have been provided with a hint on how to approach this problem.

```
Start by investigating `data/example.csv` and
`src/database/createTables/create-tables.sql`.
Then, run tests and try to fix them one-by-one.
The `tests/importExport.test.ts` file should be
left for the end as it tests importing and exporting
together with a database and a spreadsheet.
```

After looking at the SQL file, we can see that it contains a list of `CREATE TABLE` statements that model the following entities:

- address
- customer
- driver
- invoice
- package

If we look through `importCsv.ts`, we will find our data import flow:

```
const parsed = await parseCsv(spreadsheetSchema, csvContent)
const wanted = formRows(parsed)

syncTables(database, wanted)
```

By checking the input/output of each function, we can see that the data import flow consists of 3 steps:

```
- parsing CSV data (string -> array of objects)
- mapping objects to database rows
```

```
    (array of spreadsheet format objects -> array of database format objects)
  - performing SQL queries (array of database format objects -> database)
```

2. Run tests.

If we run `npm test` (same as `npm run test`, which is same as `npx vitest`), we will find out that we have some passing tests and some failing tests. Here are the failing tests:

```
❯ tests/importExport.test.ts (1)
    × preserves the same file output as the input used for creating a database
  ❯ src/importSpreadsheet/tests/importCsv.spec.ts (1)
    × should save given records to a database
  ❯ src/importSpreadsheet/syncTables/syncTables.spec.ts (3)
    × should save a given list of addresses to a database
    × should overwrite existing addresses
    × should sync multiple tables
  ❯ src/exportSpreadsheet/tests/generateCsv.spec.ts (1)
    × should return all results in the CSV format
```

Where should we start? In general, the highest level tests are good for understanding the problem and making sure that we are on the right track. If we check that test out, we would see that by the end of the exercise, we should be able to import a CSV file into a database and then export what's in the database back to a CSV file. At the end, we should end up with a CSV file that is identical to the one we started with.

While this test is a good place to understand our goal, it is not a good place to start. It is a complex test that tests multiple things at once. If we start with this test, we will need to implement a lot of code before we can even run it. This will make it hard to know if we are on the right track.

Note: `importExport.test.ts` is not colocated with the rest of the tests as it does not test a single module, but rather the whole system. It is a good idea to leave it for the end.

3. Allow inserting data into the database from a CSV file.

We can start either with import or export part. It does not matter much which one we start with.

Let's start with import as it is the first step in our data flow.

For import to work, we need to get the following tests to pass:

```
❯ syncTables.spec.ts (3)
```

```
 ❯ importCsv.spec.ts (1)
```

Let's start with `syncTables.spec.ts` as it is a "deeper" test testing only the `syncTables` module, while `importCsv.spec.ts` tests a higher level module that uses `syncTables` internally.

It contains 3 tests. Let's start with the first one by adding a `.only` modifier to the `it` function:

```
it('should save a given list of addresses to a database', async () => {
  const db = await createTestDatabaseWithTables()
  const rowsByTable = {
    address: [addressJohn, addressJane],
  }

  syncTables(db, rowsByTable)

  const addresses = db.prepare('SELECT * FROM address').all()
  expect(addresses).toHaveLength(2)
})
```

From the setup part, it is clear that the first function creates a database instance with tables. If we check the source code, we see that the function creates a new SQLite3 database in memory and then runs a SQL script that creates the tables. An in-memory database allows us to run our tests without having to worry about the database state in our machine - the database will not be persisted to the disk and it will not affect other tests. This is a good practice in testing as we want our tests to be isolated from each other.

In the test, we see that the function accepts a database instance and some sort of { [tableName]: rows } object. Given the expectation, we can assume that the function will insert the data from the `rowsByTable` object into the database.

We could start out with the most hard-coded implementation:

```
/* syncTables/index.ts */
export default (database, rowsByTable) => {
  database
    .exec(`
      INSERT INTO address (id, customer_id, name, address_full)
      VALUES
        (1, 15, 'John Doe', '123 Main St'),
        (2, 15, 'Jane Doe', '456 Main St')
      `)
}
```

That should get things going. Here we are completely ignoring the input (`rowsByTable`) and just inserting a single row into the database. While we will not be able to use this moving forward, it is a good starting point to make sure that we can insert data into the database. This works as a **"sanity check"** to make sure that we are not missing anything obvious.

We can now refactor our function in a form of parameterized SQL query where we use `?` as a placeholder for the values:

```
database
  .prepare(`
    INSERT INTO address (id, customer_id, name, address_full)
    VALUES (?, ?, ?, ?), (?, ?, ?, ?)
  `)
  .run([
    1, 15, 'John Doe', '123 Main St',
    2, 15, 'Jane Doe', '456 Main St',
  ])
```

This is a heavily recommended practice to make our queries safe from SQL injection.

At this point, we are still ignoring the provided `rowsByTable` object. However, we are slowly moving towards a solution that will work for multiple rows all the while making sure that our first test is passing.

We can now refactor our solution to make it more generic. Each problem can be addressed separately:

1. we can use some sort of iteration to insert multiple rows. We can form an `INSERT INTO address (id, name, customer_id, address_full) VALUES ${values}` query.
2. We could either repeat the same logic for each table or we could try to make it generic by having a function which would accept a table name, list of columns, values and would run the `INSERT INTO ${table} (${columns}) VALUES ${values}` . We could then call that function for each table. Our 2nd test `it('should insert data into multiple tables')` will help us to test this functionality.

Note on validation: in general, we want a step (or multiple steps) in our data pipeline that ensure that the data is correct. For example, we could validate the data before inserting it into the database. However, for the sake of keeping this exercise focused on SQL, validation has already been implemented in the `parseFile` module and its match to our database has been enforced through the `formRows` module. This is a good example of how we can split our problem into multiple modules and how we can make sure that each module is responsible for a single thing.

4. Allow updating existing data in the database.

When we are done with the first two test, we can move on to the third test:

```
it('should overwrite existing addresses', async () => {
  // ...
})
```

We can implement this in 2 ways:

- by selecting existing data from the database by id and then updating the data that we have in the CSV file
- by using `ON CONFLICT` clause in the `INSERT` statement. This would allow us to perform INSERT and UPDATE in a single statement. This is a good option when we can know a unique/primary key of the data we are inserting in advance. In our case, all our packages have an `id` field, which is a primary key.

In the provided solution, we went with the second option. However, both options are valid for a problem like this.

5. Allow exporting data to a CSV format.

Exporting data to a CSV format would require doing the opposite of what we did when importing data from a CSV file. We would need to:

- read data from the database (`SELECT` statement)
- map the data to the JS object for the spreadsheet
- convert the data to CSV

If we check the `exportSpreadsheet/index.ts` , and then subsequently `exportSpreadsheet/generateCsv.ts` , we can see the final part of the flow has been already implemented. We can now focus on the first two parts.

By running our tests on the import part, we find that our `queryDatabase.ts` module is not returning anything useful.

This part focuses on performing a multi-table `SELECT` statement with a few `JOIN` statements to save us from having to perform multiple queries.

In our case, given the requirements, we know that all our rows will include a `package` . This also excludes us from using a `RIGHT JOIN` as we always want to include the `package` table in our result.

We can then join other tables to `package` . What type of join should we use? Given that in some cases, packages might not have an assigned driver, invoice or a delivery address, we should use `LEFT JOIN` to make sure that we are not excluding any packages from our result. After listing our selects and joins we end up with the query in the provided solution (`queryDatabase.ts` file).

6. Run all tests to make sure we are not breaking anything.

Run all tests, including your integration test - `importExport.test.ts` . If it fails, while all other tests pass:

- isolate in which step it fails and which module is responsible for that step
- add a test for that module that should cover the failing case
- fix the module while running its unit test
- run the integration test again

7. Refactor your code to make it more readable and independent from implementation details.

Finally, we can refactor our code to make it more readable and independent from implementation details. For example, we could:

- add missing types
- add more documentation
- rename variables and functions to make the code more readable
- consider consolidating repeated code into functions or classes
- think about making our solution withstand possible future changes - change in order of columns in the CSV file, change in the database schema, etc.

Given that we have a test suite, we can refactor our code with confidence that we are not breaking anything.