TASK

Relational Databases

Visit our website

Introduction

WELCOME TO THE DESIGN AND BUILD A RELATIONAL DATABASE TASK!

In this task, we introduce the relational database. We will also discuss how to evaluate and design good table structures to control data redundancies and avoid data anomalies using the normalisation process.

WHAT IS A RELATIONAL DATABASE?

A relational database organises data as a set of formally described tables. Data can then be accessed or reassembled from these tables in many different ways without having to reorganise the database tables.

To put it simply, a relational database organises data into tables and links them based on defined relationships. These relationships then enable you to retrieve and combine data from one or more tables with a single query.

Each table in a relational database has a unique name and may relate to one or more other tables in the database through common values. These tables, or relations as they are sometimes called, contain rows (records) and columns (fields). Each row contains a unique instance of data for the categories defined by the columns. For example, a table that describes a customer can have columns for name, address, phone number, and so on. Rows are sometimes referred to as tuples and columns as attributes.

A relationship is a link between two tables. Relationships make it possible to find data in one table that pertains to a specific record in another table.

Tables in a relational database often contain a **primary key**, which is a column or group of columns used as a unique identifier for each row in the table. For example, a customer table might have a column called CustomerID that is unique for every row. This makes it easy to keep track of a record over time and to associate a record with records in other tables.

Tables may also contain **foreign keys**, which are columns that link to primary key columns in *other* tables, thereby creating a relationship. For example, the Customers table might have a column called SalesRep that links to EmployeeID, which is the primary key in the Employees table. In this case, the SalesRep column is a foreign key. It is being used to show that there is a relationship between a specific customer and a specific sales rep.

A **relational database management system (RDBMS)** is the software used for creating, manipulating, and administering a database. The RDBMS is often referred to as the database. Many commercial RDBMSs use **Structured Query Language (SQL)**. SQL queries are the standard way to access data from a relational database. SQL queries can be used to create, modify, and delete tables, as well as select, insert, and delete data from existing tables.

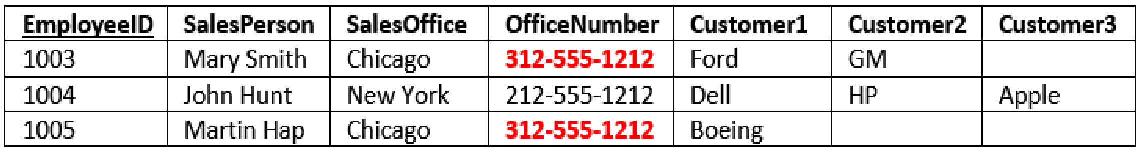
DATA REDUNDANCY

When talking about databases, it is important to know about the common database or data organisation issues and how to deal with them.

Data redundancy is a condition created within a database or data storage technology in which the same piece of data is held in two separate places. As it is unlikely that data stored in different locations will always be updated consistently, duplication of data can create 'islands of information' (the term given for the scattered data locations) that can contain different versions of the same data (for example, two different fields within a single database).

Uncontrolled data redundancy can cause issues such as:

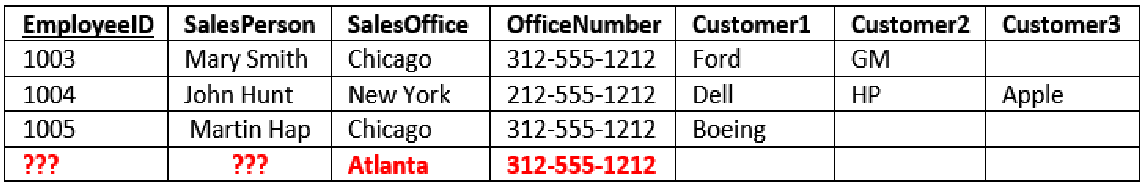
- **Data inconsistency:** This exists when different and conflicting versions of the same data appear in different places.
- **Poor data security:** Having multiple copies of data increases the chances of a copy of the data being accessed without authorisation.
- **Data anomalies:** An anomaly is an abnormality. Ideally, a field value change should be made in only a single place. A data anomaly develops when all of the required changes in the redundant data are not made successfully. Data anomalies are defined by Coronel and Morris (2014) as:
    - *Update anomalies:* Occurs when the same information is recorded in multiple rows. For example, in an Employee table, if the office number changes, then there are multiple updates that need to be made. If these updates are not successfully completed across all rows, then an inconsistency occurs.



EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	**312-555-1212**	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	**312-555-1212**	Boeing		

(Source: Sharma & Kaushik, 2015)

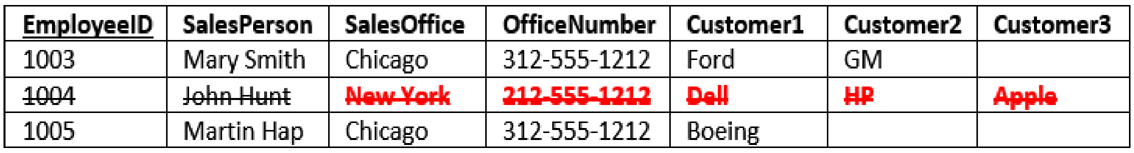
-
    - *Insertion anomalies:* There is data we cannot record until we know the information for the entire row. For example, we cannot record a new sales office until we also know the salesperson because, in order to create the record, we need to provide a primary key. In our case, this is the EmployeeID.



EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		
???	???	Atlanta	312-555-1212			

(Source: Sharma & Kaushik, 2015)

- *Deletion anomalies:* Deletion of a row can cause more than one set of facts to be removed. For example, if John Hunt retires, then deleting that row will cause us to lose information about the New York office.



EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
~~1004~~	~~John Hunt~~	~~New York~~	~~212-555-1212~~	~~Dell~~	~~HP~~	~~Apple~~
1005	Martin Hap	Chicago	312-555-1212	Boeing		

(Source: Sharma & Kaushik, 2015)

NORMALISATION

Normalisation is a process for evaluating and correcting table structures to minimise data redundancies and therefore reduce the likelihood of data anomalies. In other words, normalisation is a method to remove anomalies and bring the database to a consistent state.

Normalisation works through a series of stages called **normal forms**. The first 3 stages are described as the first normal form (1NF), second normal form (2NF), and third normal form (3NF). 2NF is structurally better than 1NF, and 3NF is structurally better than 2NF. Normally, 3NF is as high as you need to go in the normalisation process for most business database design purposes.

The objective of normalisation, according to Rob, Coronel, and Crockett (2008), is to create tables that have the following characteristics:

- Each table represents a single subject. For example, an employee table will only contain data that directly pertains to employees.
- No data item will be unnecessarily stored in more than one table so that data is only updated in one place.
- All attributes in a table are dependent on the primary key.

Conversion to first normal form

According to Coronel & Morris (2014), normalisation can be done with a simple three-step procedure.

- ***Step 1: Eliminate the repeating groups***

The data must be presented in a tabular format, where each cell has a single value and there are no repeating groups. A repeating group derives its name from the fact that multiple entries of the same type can exist for any single key of an attribute key. These entries, or repeating groups, will have identical structures but may consist of several fields.

PROJ_NUM	PROJ_NAME	EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR	HOURS
15	Evergreen	103	June Arbaugh	Elect. Engineer	$67.55	23
		101	John News	Database Designer	$82.00	19
		105	Alice Johnson	Database Designer	$82.00	35
		106	William Smithfield	Programmer	$26.66	12
		102	David Senior	System Analyst	$76.43	12
18	Amberwave	114	Ann Jones	Applications Designer	$38.00	24
		118	James Frommer	General Support	$14.50	45
		104	Anne Remoras	System Analyst	$76.43	32
		112	Darlene Smithson	DSS Analyst	$36.30	44

(Source: Coronel & Morris, 2014)

Take a look at the table above. Note that each project number can reference a group of related data entries. The Evergreen project, for example, is associated with five entries, one for each person working on the project. Those entries are related because each of them has a value of 15 for `PROJ_NUM`. The number of entries in the repeating group grows by one each time a new record is entered for another person who is working on the Evergreen project.

To eliminate repeating groups, you need to eliminate the **null values** (a null value is used in databases to signify a missing or unknown value) by making sure that each repeating group attribute contains an appropriate data value. Doing this converts the table above to 1NF like the one below.

PROJ_NUM	PROJ_NAME	EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR	HOURS
15	Evergreen	103	June Arbaugh	Elect. Engineer	$67.55	23
15	Evergreen	101	John News	Database Designer	$82.00	19
15	Evergreen	105	Alice Johnson	Database Designer	$82.00	35
15	Evergreen	106	William Smithfield	Programmer	$26.66	12
15	Evergreen	102	David Senior	System Analyst	$76.43	12
18	Amberwave	114	Ann Jones	Applications Designer	$38.00	24
18	Amberwave	118	James Frommer	General Support	$14.50	45
18	Amberwave	104	Anne Remoras	System Analyst	$76.43	32
18	Amberwave	112	Darlene Smithson	DSS Analyst	$36.30	44

- ***Step 2: Identify the primary key***

  You should note that `PROJ_NUM` is not an adequate primary key, because it does not *uniquely identify* one row of the table and therefore does not identify all of the remaining entity (row) attributes. To create a primary key that will uniquely identify an attribute value, the new primary key must therefore be composed of a combination of `PROJ_NUM` and `EMP_NUM`. So, for example, if you know that `PROJ_NUM` = 15 and `EMP_NUM` = 103, the entries for the attributes can only be Evergreen, June Arbaugh, Elect. Engineer, $67.55, and 23.

- ***Step 3: Identify all dependencies***

  You have already identified the following dependency by identifying the primary key in step 2:

  `PROJ_NUM`, `EMP_NUM` -> `PROJ_NAME`, `EMP_NAME`, `JOB_CLASS`, `CHG_HOUR`, `HOURS`

  This means that `PROJ_NAME`, `EMP_NAME`, `JOB_CLASS`, `CHG_HOUR`, and `HOURS` are dependent on the combination of `PROJ_NUM` and `EMP_NUM`. There are other dependencies, however.

  For example, the project number determines the project name. You can write this dependency as:

`PROJ_NUM` -> `PROJ_NAME`

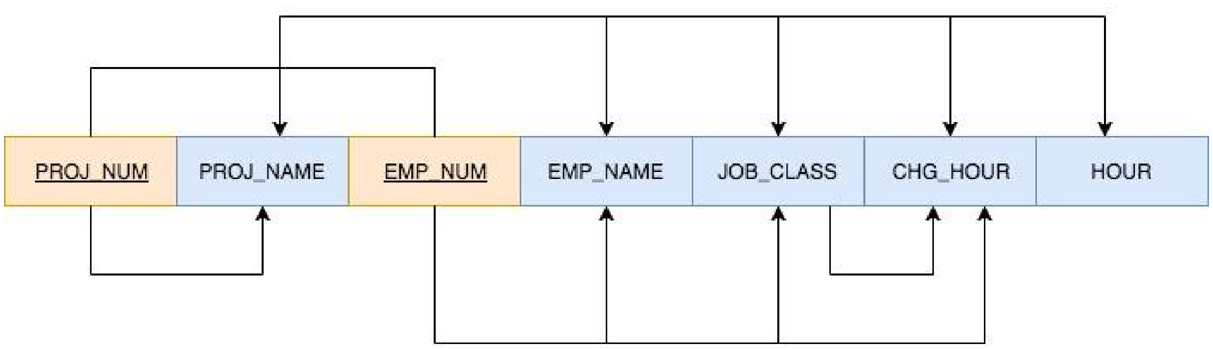
You can also determine an employee's name, job classification, and charge per hour if you know an employee number. You can write this dependency as:

`EMP_NUM` -> `EMP_NAME`, `JOB_CLASS`, `CHG_HOUR`

Lastly, knowing the job classification means knowing the charge per hour for that job classification:

`JOB_CLASS` -> `CHG_HOUR`

A dependency diagram can help depict dependencies. They are helpful in getting a bird's eye view of all the relationships among a table's attributes.



(Source: Coronel & Morris, 2014, p. 198)

Look at the dependency diagram above, and note the following:

1. The primary key attributes are underlined and shaded in a different colour.
2. The arrows above the attributes indicate all desirable dependencies. Desirable dependencies are those based on the primary key. Note that the entity's attributes are dependent on the combination of `PROJ_NUM` and `EMP_NUM`.
3. The arrows below the diagram indicate less desirable dependencies. There are two types of less desirable dependencies:
    a. Partial dependencies, which are dependencies based on only a part of the composite primary key. For example, you only need to know `PROJ_NUM` to determine `PROJ_NAME`.

    b. Transitive dependencies, which are dependencies of one non-prime attribute (an attribute that is not part of a primary key) on another non-prime attribute. For example, `CHG_HOUR` is dependent on `JOB_CLASS`.

A table is in 1NF when:

- all of the key attributes are defined;
- there are no repeating groups in the table; and
- all attributes are dependent on the primary key (Coronel & Morris, 2014).

Conversion to second normal form

The relational database design can be improved by converting the database into a format known as the second normal form. According to Coronel & Morris, the following steps are required to convert a database to second normal form:

- ***Step 1: Write each key component on a separate line***

  Write each key component on a separate line, then write the original (composite) key on the last line:

  ```
  PROJ_NUM
  EMP_NUM
  PROJ_NUMEMP_NUM
  ```

  Each component will become the key in a new table. In other words, the original table is now divided into three tables (`PROJECT`, `EMPLOYEE`, and `ASSIGNMENT`).

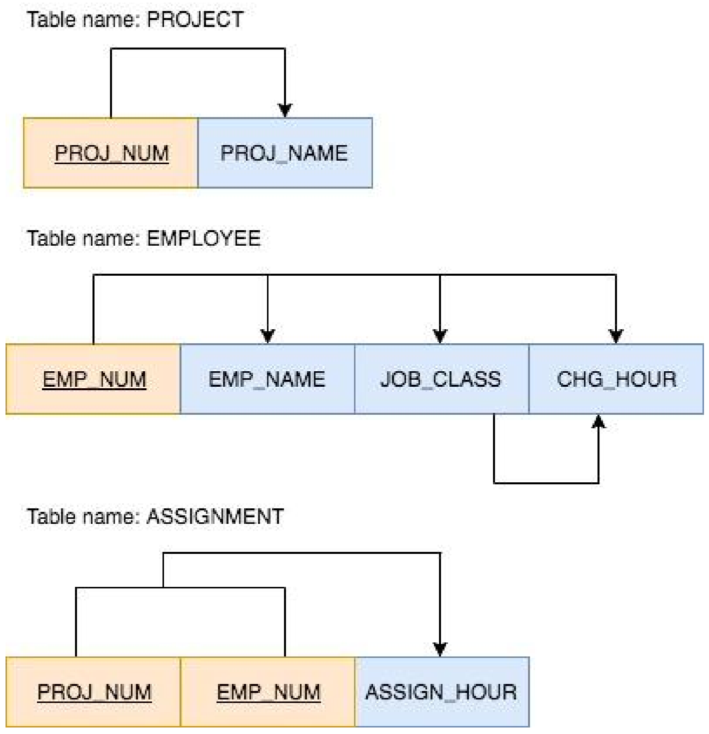
- ***Step 2: Assign corresponding dependent attributes***

  Then use the dependency diagram above to determine those attributes that are dependent on other attributes. The dependencies for the original key components are found by examining the arrows below the dependency diagram. The three new tables are described by the following relational schemas:

  ```
  PROJECT (PROJ_NUM, PROJ_NAME)
  EMPLOYEE (EMP_NUM, EMP_NAME, JOB_CLASS, CHG_HOUR)
  ASSIGNMENT (PROJ_NUM, EMP_NUM, ASSIGN_HOURS)
  ```

  The number of hours spent on each project by each employee is dependent on both `PROJ_NUM` and `EMP_NUM` in the `ASSIGNMENT` table; therefore, you place those hours in the `ASSIGNMENT` table as `ASSIGN_HOURS`.

The dependency diagram below shows the result of Steps 1 and 2.



At this point, most of the anomalies discussed earlier have been eliminated. For example, if you now want to add, change, or delete a `PROJECT` record, you need to only go to the `PROJECT` table and add, change, or delete only one row.

A table is in 2NF when:

- it is in 1NF; and
- it includes no partial dependencies; that is, no attribute is dependent on only a portion of the primary key.

It is still possible for a table in 2NF to exhibit transitive dependency; that is, one or more attributes may be functionally dependent on non-key attributes (Coronel & Morris, 2014).

Conversion to third normal form

The data anomalies created by the database organisation shown above are easily eliminated by completing the following steps as outlined by Rob et al. (2008):

- ***Step 1: Identify each new determinant***

  For every transitive dependency, write its determinant as a PK for a new table. A determinant is any attribute whose value determines other values within a row. You will have three different determinants if you have three different transitive dependencies. The dependency diagram from earlier shows a table that contains only one transitive dependency. Therefore, we write the determinate for this transitive dependency as:

`JOB_CLASS`

- ***Step 2: Identify the dependent attributes***

  Identify the attributes that are dependent on each determinant identified in Step 1 and identify the dependency. You write in this case:

  `JOB_CLASS` -> `CHG_HOUR`

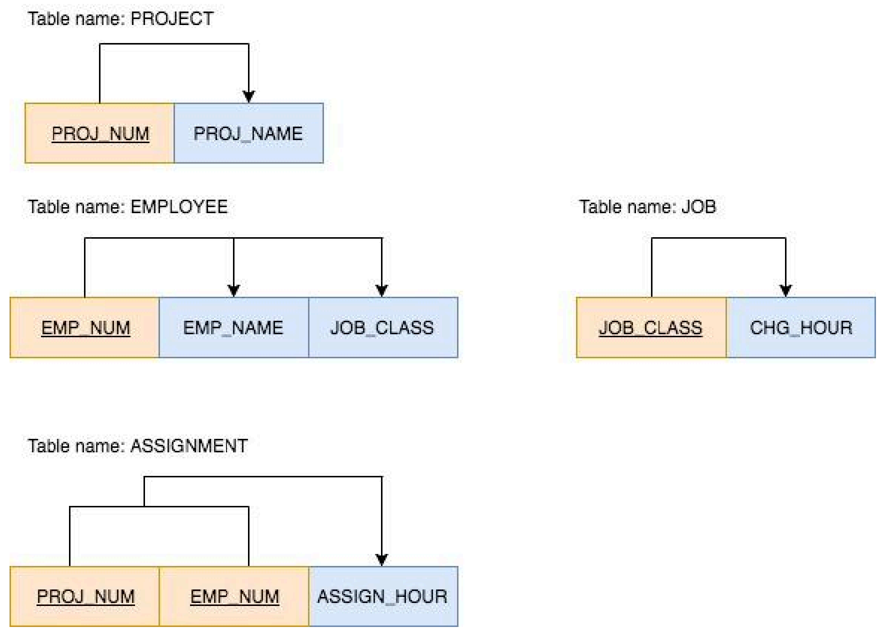
  Give the table a name that reflects its contents and function. We shall name this table **`JOB`**.

- ***Step 3: Remove the dependent attributes from transitive dependencies***

  Eliminate all dependent attributes in the transitive relationship from each of the tables that have such a relationship. In this example, we eliminate `CHG_HOUR` from the **`EMPLOYEE`** table shown in the dependency diagram above to leave the **`EMPLOYEE`** table dependency definition as:

  `EMP_NUM` -> `EMP_NAME`, `JOB_CLASS`

  Notice that the **`JOB_CLASS`** remains in the **`EMPLOYEE`** table to serve as the foreign key. You can now draw a new dependency diagram to show all of the tables you have defined in the steps above. Then check the new tables as well as the tables you modified in Step 3 to make sure that each table has a determinant and that no table contains inappropriate dependencies. The new dependency diagram should look as follows:



(Source: Rob, Coronel, & Crockett, 2008, p. 258)

The above dependency diagram is what is created after completing Steps 1–3. After the conversion has been completed, your database should contain four tables:

```
PROJECT (PROJ_NUM, PROJ_NAME)
EMPLOYEE (EMP_NUM, EMP_NAME, JOB_CLASS)
JOB (JOB_CLASS, CHG_HOUR)
ASSIGNMENT (PROJ_NUM, EMP_NUM, ASSIGN_HOURS)
```

This conversation has eliminated the original `EMPLOYEE` table's transitive dependency. The tables are now said to be in third normal form (3NF).

A table is said to be in 3NF when:

- it is in 2NF; and
- it contains no transitive dependencies (Rob et al., 2008).

CROSS-QUERYING A NORMALISED DATABASE

Data redundancy is great! Sadly, it does result in slightly more complexities in our queries. Now, instead of just looking in one table, we have to start looking in two (or more!) tables. On the bright side, this is just as intuitive as regular lookups!

Introducing the `JOIN` operator: this is the glue that holds multiple tables together. There are five types of joins:

- `INNER`
- `LEFT OUTER`
- `RIGHT OUTER`
- `FULL OUTER`
- `CROSS`

Wow, that's a lot to remember! However, we don't need to worry too much about it: Only the `INNER` and `LEFT OUTER` joins are considered as the most common joins.

Let's expand a bit more on these types of joins.

`INNER` joins are probably the most common overall. The `INNER` join requires a column between two tables that will have the same values. Each row in the result of the query will contain columns from both tables. These rows correspond to rows in the joined tables where the specified common column matches. These are particularly useful in using foreign keys.

Let's say that you have two tables: `Employee` and `Department`. There is a foreign key in `Employee` showing which department it belongs to. You have been tasked with listing all employee names in the HR department. The department name is in the Department table, and the employee name is in the Employee table. This can be fixed with the `INNER` join like this:

```
SELECT Employee.name -- To access from a specific table, it is
Table.column_name
FROM Employee INNER JOIN Department
ON Employee.dept_id = Department.dept_id -- matching column
WHERE Department.dept_name = "HR"; -- filters according to criterion
```

Note the two important keywords: `INNER JOIN` and `ON`. `INNER JOIN` specifies that you want to use the `INNER JOIN`, and `ON` is where you specify the matching column values.

`LEFT OUTER` joins are similar to `INNER` joins. Like the `INNER` join, it will use a specified column to match values across two tables. There is just one small distinction. Let's say that you are using a `LEFT OUTER` join on Table A and Table B. Column X is the common column between the two tables. If there is a value in X for Table A that doesn't exist in Table B, it will still show in the output (but with rows in the Table B columns just showing null values).

Extra resource

If you'd like to know more about the interesting field of databases, we recommend reading the book **Database Design (2nd ed.)** by Adrienne Watt. **Chapters 11 and 12** of this book are a great supplement to this task.

Take note:

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation.

Give it your best attempt and submit it when you are ready.

You will receive a 100% pass grade once you've submitted the task.

When you submit the task, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for this task, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you've done that, feel free to progress to the next task.

Auto-graded Task 1

Answer the following questions:

- What is normalisation?
- When is a table in 1NF?
- When is a table in 2NF?
- When is a table in 3NF?
- What is a foreign key?

Auto-graded Task 2

Create a database file called `School.db`. This will contain two tables: `Student` and `Course`. The `Student` table will have a foreign key to the `Course` table. The `Course` table contains the following attributes:

- `course_code` – a 5-character primary key
- `course_name` – the name of the course
- `course_description` – a description of the course
- `teacher_name` – the name of the person who is teaching the course
- `course_level` – a number showing the level of the course (either 1, 2, or 3)

Place the tables in **`School.db`** by running the **`create_course_table.sql`** and **`create_student_table.sql`** scripts, in that order. Create the following files and place the respective query in each file. Please note that each query must either use an `INNER JOIN` or a `LEFT OUTER JOIN` – queries that do not contain the `JOIN` keyword will be marked as incorrect.

- **`machine_learning.sql`** – List the *names* and *surnames* of all students doing the DS03 course.
- **`final_stage.sql`** – List the *email addresses* of all students who are doing a level 3 course.

- **`easiest_courses.sql`** – List the *first names* of all students that achieve a mark of 70 or above, along with the *course name* that they got a mark of 70 or above in.
- **`julia_python.sql`** – List the *marks* of all students who have been taught by Julia Python.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved? Do you think we've done a good job?

Click here to share your thoughts anonymously.

References:

Coronel, C., & Morris, S. (2014). *Data systems: Design, implementation, and management*. Cengage Learning.

Rob, P., Coronel, C., & Crockett, K. (2008). *Database systems: Design, implementation, and management*. Cengage Learning.

Sharma, R., & Kaushik, S. (2015). *Database management system*. Horizon Books.